Bit-Serial Reed-Solomon Decoders in VLSI

Thesis by

Douglas L. Whiting

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

5157:TR:84

1984

(Submitted August 22, 1984)

BIT-SERIAL REED-SOLOMON DECODERS IN VLSI

Acknowledgements

Because a large fraction of my adult life has been spent as a student at Caltech, a complete list of the people who have aided my academic progress is much too long to be included here. However, a few individuals deserve special mention. My parents gave me financial assistance for years as well as constant encouragement and support. It was the enthusiastic teaching of Bob McEliece that first attracted me to this area of research, and his insights and interest have kept me on the right road ever since. Cal Jackson has spent countless hours customizing TEX to my often exacting whims; his suggestions, comments, and incredibly painstaking proofreading have been invaluable. Many thanks to Chuck Seitz, who was a helpful adviser through my Master's Thesis research. Bob Anderson, Anne-Marie Brest, Neil Brock, and Brenda Roder all worked valiantly on the decoder floor plan and the cell layout, showing that a decoder chip is possible, although not easy. The friendship of Gary Clow, John Tanner, John Ngai, and many others has made school an enjoyable place to be, and the proofreading and advice of Bill Dally have been particularly appreciated over the past few months. But most of all, I am grateful to Dee, Claire, and Wendy, for their constant love and for the joy that having a family brings to my life.

This work was sponsored in part by the Defense Advanced Research Agency, ARPA Order #3771, and monitored by the Office of Naval Research, Contract #N00014-79-0597, and by a National Science Foundation fellowship.

Abstract

Reed-Solomon codes are known to provide excellent error-correcting capabilities on many types of communication channels. Although efficient decoding algorithms have been known for over fifteen years, currently available decoder systems are large both in size and in power consumption. Such systems typically use a single, very fast, fully parallel finite-field multiplier in a sequential architecture. Thus, more processing time is required as the code redundancy increases. By using many arithmetic units on a single chip, it is possible to exploit the concurrency inherent in the decoding algorithms to attain performance levels previously possible only with large ECL systems.

An investigation into the structure of binary extension fields reveals that the common arithmetic operations used in decoding can be implemented quite efficiently in a bit-serial fashion, using any of several bases over GF(2). Berlekamp's dual-basis multiplier is generalized to the product of two arbitrary field elements, and a necessary and sufficient condition is then derived for the existence of a self-dual basis. The efficient methods presented for bit-serial multiplicative inversion greatly reduce the complexity traditionally associated with this operation.

Using these bit-serial techniques, several architectures for implementing each phase of the known Reed-Solomon decoding algorithms are presented and compared. Simple methods are presented to allow power-sum syndrome decoders to handle codes with a variety of block lengths and redundancies. Each approach comes within a factor of $\log n$ (where n is the block length of the code) of the recently derived asymptotic lower bounds for both time and area. Results from a student project to lay out a prototype decoder chip using the Berlekamp-Massey algorithm are also discussed. By utilizing the parallelism inherent in the key equation solution, these architectures can decode received words at a speed independent of the redundancy of the code.

Contents

Figures and Tables

Chapter 1

Introduction

1.1 Reliable Digital Communication

Communication can be defined as the transfer of information from one point to another over a channel or medium. Because the channel separates source and destination points in either space or time (and often both), there is a finite probability that the information received will differ from the transmitted data. Although all information is ultimately transmitted in an analog form over the channel, this thesis will be concerned exclusively with digital data, i.e., systems in which analog values are quantized at some point in the communication process. Such quantization allows rigorous analysis using discrete mathematics. For example, in the paper that gave birth to the science of information theory, Shannon showed that, by adding properly chosen redundant information to the data transmitted, communication can be made arbitrarily reliable [46]. The required amount and complexity of this redundancy obviously depends on the channel: in general, as the medium becomes more sensitive to noise or interference, more overhead is required.

To apply Shannon's theorem in practice, an engineer must select an acceptable error probability and design the system to that specification. There are several basic approaches to achieving a given level of reliability in communication. First, the designer can attempt to improve the channel characteristics by employing more sophisticated modulation/demodulation schemes, more powerful transmitters, more sensitive receivers, etc. On the other hand, using coding theory, redundant information can be added to provide error detection and/or correction capabilities. One very powerful method of increasing reliability by using coding techniques is to request a retransmission when errors are detected [2]. However, such a scheme presupposes that retransmission is possible (which will not always be the case) and that the probability of error-free transmission is acceptably high. This approach will not be studied here, although it can be very effectively coupled with error correction. Instead, the coding technique of interest here involves the use of *error-correcting codes* (ECC) to both detect and correct any errors which have occurred in transmission. Each of these approaches will inevitably increase the system cost, whether measured in physical size, power consumption, data latency, or development dollars. However, all of these methods can be important, and it is clear that some mixture of coding and channel improvements will minimize a given cost metric in meeting the reliability objective.

One interpretation of Shannon's theorem is that it is an inefficient use of resources to build a channel that is overly reliable. However, until recently, the cost of implementing powerful error-control codes in a communication system has been prohibitively high, and unfortunately this cost barrier often stems as much from a lack of appreciation of Shannon's theorem and coding principles as from the actual ECC hardware. Thus, in all but the highest performance or cost-insensitive applications, the tradeoff has usually been to improve the channel characteristics, employing only relatively unsophisticated coding techniques (if any). The advent of VLSI offers the promise of incorporating the entire coding system onto a single integrated circuit. Encapsulation of the low-level coding details into an inexpensive chip will allow an engineer to make more intelligent decisions about the tradeoffs involved in reliable system design. For example, in magnetic data storage, the proper compromise may well be to employ ECC and use less sophisticated read/write electronics with a higher bit density, thus providing higher capacity and lower cost storage that is more reliable despite the higher raw error rate, as compared to a conventional system. The purpose of this thesis is to propose a set of architectures and arithmetic techniques that will allow one particular type of coding system to be integrated onto a chip.

1.2 Conventional Reed-Solomon Decoder Systems

Although the need for reliability in communication is universally recognized, the criterion for acceptable data integrity varies widely from application to application. Typically, communication systems such as telephone or digital audio in which the final output is an analog value can allow a much higher error rate than channels where the output is to be further processed in its digital form. For example, a bit error rate of 10^{-6} is generally acceptable for the telephone switching network, while such a figure would be catastrophic in a digital magnetic storage system. Further, raw error statistics for channels of interest are even more diverse. In some cases, errors are roughly independent from bit to bit, while many media are characterized by bursts of errors; often, soft (analog) information extracted from the demodulator can aid tremendously in reconstructing the corrected data, but on other channels a quantized (hard) output is all that is realistically of help.

Such diversity in the error statistics and the reliability requirements of different channels obviously implies that no single error-correcting code can meet the needs of every application. However, Reed-Solomon codes are among the most versatile and powerful codes available, with an inherent capability of correcting both random and burst errors and of incorporating a limited amount of soft decision information. Thus, Reed-Solomon (RS) codes are a prime candidate for VLSI implementation. Efficient RS encoder design is relatively well understood [5], but the problem of optimal decoder architectures for VLSI remains an open question. The chapters of this thesis lead up to an efficient set of methods for implementing both the finite-field arithmetic and the decoding algorithms, utilizing as much parallelism as possible.

Before beginning our investigation, it will be helpful to review the architectures of some RS decoders that have actually been implemented using conventional digital logic as a point of comparison. Cohen, in his doctoral thesis, examines three different RS decoder systems in terms of performance and chip count [19]; his results are briefly summarized in the first three rows of Table 1-1. The last row of this table is somewhat of an extrapolation of the previous data to the next generation of RS decoders. In Chapter three we will explain the significance of the numbers in the column labeled 'Code,' but for our purposes here it suffices to know that all the decoders work over the same field, have the same block length, and can correct roughly the same number of errors. Thus, it makes sense to compare the performance of these systems. The chip count and clock rate give some indication of the hardware complexity of each decoder,

and it should be noted that these numbers will vary from implementation to implementation. The final performance measure is throughput, in terms of decoded bits output per second when the maximum number of errors must be corrected. Again, these figures are subject to variation depending on the particular system configuration; in general, it is possible to trade hardware for speed.

Processor Type	Chip Count	Clock Rate	Code	Throughput
8086 microprocessor	25	6MHz	(255,243)	20Kb/sec
2901 bit-slice	30	4MHz	(255,239)	200Kb/sec
microcoded GF1	70	16MHz	(255,239)	3Mb/sec
?? (parallel)	1	20MHz	(255,239)	20Mb/sec

Table 1-1. RS Decoder Performance

Cohen's first decoder uses a standard microprocessor, with the critical loops programmed in assembly code and the rest of the software compiled from a high-level language. Such a system is obviously fairly small and inexpensive; however, a large performance overhead is paid for the instruction fetch and decode time, not to mention the inefficiency of performing finite-field arithmetic using integer operations. The 2901 bit-slice decoder overcomes some of these limitations by providing a log/antilog table that is easily accessible to the datapath, thus allowing field multiplication to be performed in a few microcycles. Also, the micro-instruction fetch is executed in parallel with the previous micro-operation, and instruction decode time is virtually eliminated by this pipeline. However, the microcode width chosen for this decoder does not allow a branch to occur in parallel with an arithmetic operation. The GF1* architecture extends the microcode width to allow totally independent control of branching and arithmetic operations. Also, a full finite-field multiplier is included to allow very efficient implementation of the polynomial operations used most frequently in decoding, such as polynomial evaluation and convolution. For example, the GF1 processor can evaluate a polynomial of degree n in only $n+6$ microcycles. Thus, the GF1 decoder achieves high performance by very carefully matching its instruction set to the problem at hand. It is clear that a combination of these area-time tradeoffs can be chosen to produce a decoder that minimizes the cost function for a given performance objective.

One characteristic of these systems is that if only a few (namely, less than three) errors occur in a block, it is possible to optimize the decoder software to take advantage of this fact. Such a technique is often referred to as average-case optimization. If the probability of more errors is small, by using elastic buffers the system can achieve an average throughput rate significantly higher than the worst-case figures given in Table 1-1. Also, a large percentage of the decoder time is spent in syndrome computation; as we shall see in Chapter three, these syndromes constitute a minimum set of values needed to locate and correct any errors. By casting the syndrome computation into hardware, it is easily possible to detect the case of no errors, so an average-case optimized decoder would never have to spend time on error-free data. Such an approach is but another example of an area-time tradeoff, and the particular application at hand determines whether the additional throughput justifies the increased hardware cost. Whether average-case

* GF1 is a trademark of Cyclotomics, Inc.

optimization is desirable depends on the channel chosen. However, Shannon's theorem implies that a very low average error rate constitutes inefficient use of the communication medium (assuming that coding is free); thus, because for other than a small number of errors, the decoding throughput is roughly constant [19], the worst-case performance is an important parameter of these systems.

Observe also that all of Cohen's architectures are basically identical: each machine is sequential, taking little advantage of the parallelism inherently available in the decoding algorithms. In fact, virtually all conventional RS decoders have been built around the same principal architecture, using only one finite-field arithmetic unit. As Cohen has shown, many clever tricks can be employed to boost the performance of sequential computing machines for a given application, but the techniques used to improve performance by two orders of magnitude from the 8086 decoder to the GF1 system are all well-known methods of conventional computer architecture. It can be argued that such approaches have a performance ceiling for any given implementation technology, simply because of limits on the clock cycle time. Thus, as the complexity of the problem, which in this case is measured by the number of correctable errors, increases, the throughput is guaranteed to decrease for any sequential architecture.

1.3 Parallel Computation: Two Analogies

While a decoder using a sequential processor has some of the drawbacks mentioned above, it also provides several benefits. Perhaps the most important of these advantages is flexibility: by changing the software, a variety of codes can be handled, and various higher level functions (such as I/O) can be incorporated. For many applications, the disadvantages may not be of concern, while flexibility is of paramount importance. A GF1 type processor in VLSI could therefore be very useful; however, such a chip does not require any architectural innovations and thus is not of particular interest from a research standpoint. The question then arises: are there VLSI decoder architectures that provide substantially higher performance (for a given technology) without sacrificing flexibility? In the remaining chapters of this thesis several such possibilities are presented and examined. Before proceeding, however, it will be instructive to discuss two familiar examples that illustrate some of the distinctions between parallel and sequential approaches to a given computation. These examples will also provide motivation for the throughput extrapolation claimed in the last row of Table 1-1.

The first example is sorting: given a partial order denoted by $\preceq$ and a sequence of values a_i for $0 \leq i < n$, find a permutation π of the indices such that $a_{\pi(i)} \preceq a_{\pi(i+1)}$ for all i. Sorting has been exhaustively studied [1, 33], and many efficient methods are known for sorting using a sequential processor. All sequential sorting techniques involve at least $n \log n$ operations, and generally there is a tradeoff between the complexity of the algorithm and its performance. Also, if we are given partial information, such as knowing that the sequence is already almost sorted (by some metric), it may be possible to employ a less complex algorithm without suffering a performance penalty.

Now consider a parallel sorting machine of size n. How fast can the data possibly be sorted? Also, because the complexity of the architecture chosen will be directly reflected in the circuit area, what is the cost of the associated architecture? Before considering these questions, let us make a simplifying assumption. Let us suppose that the values are only made available one at a time; similarly, suppose that only one value of the result sequence can be output at a time. In other words, the values must be at least written into the machine and then read out of the machine after the sorting is complete. This assumption will almost always hold in

VLSI, especially in light of the pin count limitation on chips; so even if the values are obtained in parallel, they will have to be serialized, with time delay $O(n)$. Given such an assumption, the best we can possibly hope for is that, as soon as the last value is written into the machine, the first sorted value is available to be read from the machine. Subsequent reads should produce the remaining elements in sorted order. Such a sorter operates in linear time, $T = O(n)$, which is optimal, given our assumption.

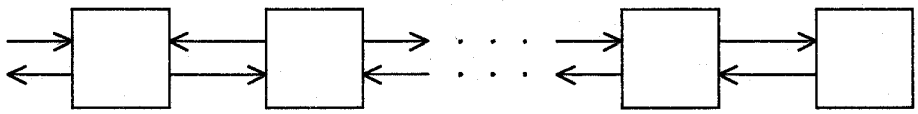

Figure 1-1. Parallel Sorting Configuration

In fact, such a machine can be built in VLSI [49]. The architecture consists of a linear array of identical cells, each of which has the ability to perform a comparison and a swap; every cell communicates only with its two neighbors. Thus, the communication required is particularly simple, as shown in Figure 1-1. This technique actually has area-time complexity $O(n^2)$, since the hardware has size $O(n)$, and the time required is also $O(n)$. The algorithm employed is essentially the bubble sort, which is inefficient on a sequential machine. However, because many of the operations of the bubble sort are executed here in parallel, such an architecture is optimal in many practial respects, particularly in light of its simple interconnection topology. Also, observe that, even given the information that the input data are in almost sorted order, it is not possible (or necessary) to improve the performance of the sorter without drastic increases in complexity.

Our second example is the discrete Fourier transform (DFT) over the complex numbers. In the following chapter the DFT is defined for finite fields, but for the sake of illustration the complex field will suffice here. In fact, we shall see in Chapter three that this example has more than a superficial relationship to the problem of decoding, because the power-sum syndromes are a subset of a DFT defined over the finite field of interest. Given a sequence of complex values a_i for $0 \leq i < n$, define the k^{th} frequency component A_k by

$$A_k = \sum_{i=0}^{n-1} \omega^{ik} a_i ,$$

where $\omega = e^{-i\frac{2\pi}{n}}$ is a primitive n^{th} root of unity. On a sequential machine, the usual complexity measurement for the DFT involves the number of multiplications, and clearly a brute force computation of the sequence A_k requires n^2 such operations. However, a more clever scheme, known as the Fast Fourier Transform (FFT), can be used to compute all frequency components in time proportional to $n \log n$ [1, 33, 48]. The FFT is particularly efficient for digital computation when n is a power of two. There are many variations of the FFT, but each approach takes advantage of the relationships between various powers of ω to transform from the time-domain values a_i to the frequency-domain vector A_k in $\log n$ iterations, each of size n.

The FFT is characterized by butterfly patterns as shown in Figure 1-2, which illustrates the data flow of a decimation-in-time FFT for $n = 8$. The points at which segments intersect

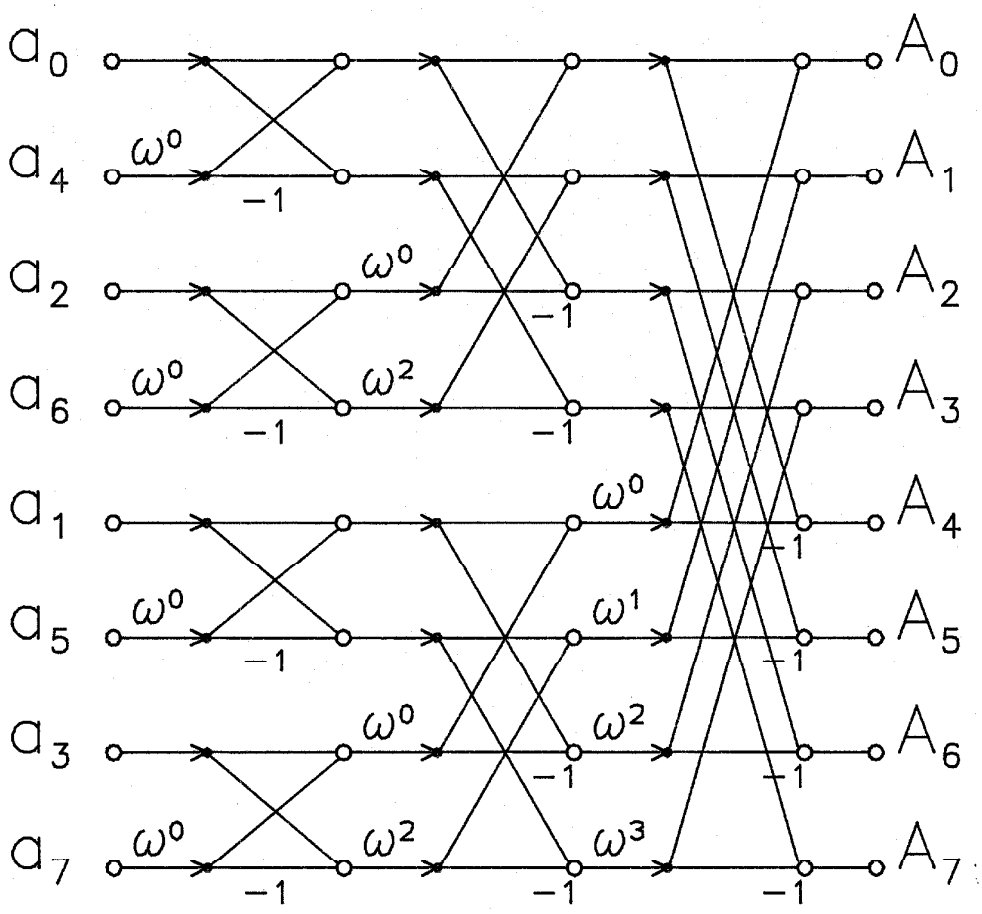

Figure 1-2. Eight-point FFT Flow Diagram

represent adders, and coefficients along a segment represent a multiplication by the constant. The $O(n \log n)$ structure of the FFT is clearly visible in this Figure. Let us now consider a parallel implementation of the DFT, with hardware complexity on the order of n. If we choose an FFT structure, so that the time complexity should be $O(\log n)$, observe that each computational node in the network has to communicate with $\log n$ other nodes, including some which are quite distant (no matter how the nodes are arranged). Further, given the same assumption we made above, to read in the data requires time n, so it appears that our solution has area-time complexity $O(n^2)$. While it is possible to implement the FFT with only hardware of size $O(\log n)$ [48], the storage required will still be $O(n)$.

Consider now a much simpler approach to DFT computation, one which will be particularly useful for coding applications, where only a fraction of the spectral components A_k need to be computed. Observe that, if we interpret the input value a_i as the coefficient of x^i in a polynomial $a(x)$, then $A_k = a(\omega^k)$. Using Horner's rule, $a(x)$ can be evaluated recursively. In other words, given the time-domain values in the order $a_{n-1}, a_{n-2}, \ldots, a_1, a_0$, let us compute an

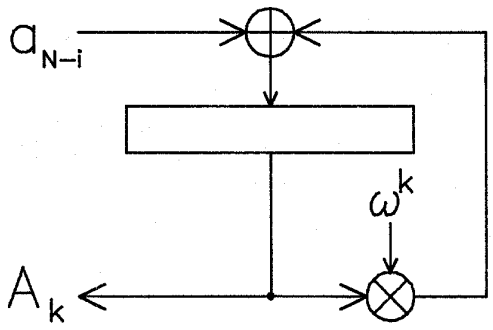

Figure 1-3. DFT Computational Node

iterative sequence of values b_i, finally producing the frequency component A_k. Set $b_0 = 0$ and define

$$b_i = \omega^k b_{i-1} + a_{n-i} \quad \text{for} \quad i = 1, 2, \ldots, n,$$

so that $A_k = b_n$. Clearly, A_k can be computed using an adder and a single multiplier involving the fixed constant ω^k, as shown in Figure 1-3. This approach to DFT computation has several advantages. First, assuming the data can be read into the system only one value at a time, as soon as the last value is input, the computation is complete; i.e., the computation time is dwarfed by I/O time. Second, each computational node is now totally independent, with no communication from node to node. As a result, if only a portion of the spectrum is required, only the nodes to calculate those values of interest need be included. Third, observe that the multiplier involved in the node which computes A_k can be specialized to the case of ω^k, which may involve considerable hardware savings. Lastly, note that n no longer has to be a power of two. Although this technique corresponds to a brute force computation of the DFT, it can be as efficient as an FFT approach in terms of area-time product, given our assumption; further, it is considerably more flexible and easier to wire, which will be extremely important for VLSI implementation.

Several conclusions about parallel computation can be drawn from these and other examples, particularly in respect to VLSI. In general, the correct measure of complexity in parallel architectures is not computation but communication. Thus, methods that are quite effective on sequential machines can be extremely difficult to implement in parallel, and vice versa. For example, as we saw in the case of sorting, often it is inefficient to optimize for average-case performance in parallel algorithms. Instead, a brute force approach, providing enough parallelism to handle the given problem, eliminates any distinction between average-case and worst-case performance and greatly simplifies the system design. Also, input and output time must be taken into consideration for VLSI simply because of pin count limitation; very often, if this time can be utilized for calculation, the I/O time will dominate the computation time, as we have seen in the previous two examples.

Our aim is to produce decoder architectures that take advantage of the parallelism available in the decoding algorithms. However, we must first lay a foundation for such discussions.

Chapter two gives a brief introduction to finite fields and the algebraic tools that are important for coding theory applications. Chapter three presents an overview of the major concepts of linear block codes, with an emphasis on Reed-Solomon codes in particular. Appendix C contains several tables which are useful for dealing with finite fields. These chapters are included as background to make the thesis self-contained and do not present any significant contributions to the field.

Several original results are presented in the remaining portions of the thesis, however. In Chapter four, efficient structures for performing finite-field arithmetic in digital logic are examined. A general expression for the transformations involved in bit-serial multiplication is derived. This expression includes Berlekamp's dual basis multiplier and Omura's normal basis multiplier as special cases, and it is shown that the dual basis multiplier is not restricted to multiplication by a constant. Next, the necessary and sufficient condition is derived for the dual basis to be identical to the canonical basis, which consists of consecutive powers of a primitive element. Also, it is shown how reciprocals over the field can be computed in an efficient bit-serial fashion; traditionally multiplicative inversion has been considered quite costly. In Chapter five, known decoding algorithms for Reed-Solomon codes are reviewed and compared. Techniques for erasure initialization in the Berlekamp and Euclidean decoding algorithms are presented which fit much more naturally into hardware than previous approaches. Using these results, in Chapter six we introduce several decoder architectures that utilize as much parallelism as possible and are suitable for VLSI implementation. In particular, for a t error correcting code over GF(2^m), these architectures have area $A = O(mt)$ and pipeline period $P = O(1)$. Our decoders have the property that the throughput in bits per second is directly related to P ; i.e., a 10 MHz clock rate implies a 10 Mbit/sec decoder throughput. Further, it is shown that a single decoder chip can handle a variety of redundancies and block lengths with little area overhead. Appendix A presents a proof that normal basis multiplication involves roughly twice the number of product terms as a dual basis multiplier. Appendix B uses the Gilbert model and presents a method of computing exact output bit error probabilities for Reed-Solomon codewords of a given blocklength, redundancy, and depth of interleaving. Previous attempts at such computations have assumed that all character errors are independent, but our method allows direct modelling of bursty channels by introducing an extra degree of freedom. These results represent a significant contribution to the techniques available for design and implementation of Reed-Solomon decoders.

Chapter 2

Finite Fields

2.1 Groups and Rings

In this chapter we will examine the structure of finite fields (also known as Galois fields, in honor of their discoverer) to lay a foundation for the results developed in the following chapters. Our purpose here is not to give an in-depth tutorial on the beautiful theory of Galois fields; many excellent texts are available which provide such an introduction on various levels of detail [4,10,29,34]. In fact, a background in complex analysis provides a good grasp of the concepts underlying finite fields; unfortunately the subject is often approached from a more theoretical point of view which can tend to obscure the relationship between Galois fields and the more familiar complex field. It is hoped that some analogies here can help bridge this gap. Also, in proving several helpful facts about functions over finite fields, we will introduce the notation used throughout the rest of this thesis.

An algebraic structure consists of a set of elements and the associated operator(s). Among the structures of interest for coding applications are groups, rings, and fields. A *group* is a set of elements G and an operator $\circ$, which satisfy the following axioms:

1. Closure: $\forall g, h \in \mathrm{G},\ g \circ h \in \mathrm{G}.$
2. Associative: $\forall f, g, h \in \mathrm{G},\ f \circ (g \circ h) = (f \circ g) \circ h.$
3. Identity: $\exists e \in \mathrm{G},$ such that $e \circ g = g \circ e = g,\ \forall g \in \mathrm{G}.$
4. Inverse: $\forall g \in \mathrm{G},\ \exists g^{-1} \in \mathrm{G}$ such that $g \circ g^{-1} = g^{-1} \circ g = e.$

Such a group is specified by $(\mathrm{G}, \circ, e)$. Familiar examples of groups are: the integers under addition, $(\mathrm{Z}, +, 0)$, the nonzero complex numbers under multiplication, $(\mathrm{C}, \times, 1)$, and the positive rationals under multiplication, $(\mathrm{Q}, \times, 1)$. Observe that, for example, addition modulo some value x also forms a group, such as $(\mathrm{Z}, + \pmod{x}, 0)$, where x is a positive integer, or $(\mathrm{R}, + \pmod{x}, 0)$, where x is a nonzero real number. Often the group operator has a physical interpretation, such as the rotations of objects in a plane, which is isomorphic to $(\mathrm{R}, + \pmod{2\pi}, 0)$. In the group axioms, it is not specified that the operator will be commutative; i.e., $g \circ h = h \circ g$ for every $g, h \in G$. For example, the set of all non-singular $n \times n$ matrices over a field (such as the reals) forms a group under matrix multiplication, but it is not commutative for $n > 1$. However, in

all the structures with which we shall be concerned, group operators will be commutative; such groups are known as commutative or *Abelian*.

A subset H of G which satisfies all of the group axioms is said to be a *subgroup*; for example, the even integers form a subgroup of the additive group of the integers. A fundamental property of finite groups, known as Lagrange's theorem, relates the size of subgroups to the parent group and will prove useful in our investigation of the multiplicative structure of finite fields.

Theorem 2.1. *(Lagrange) If G is a finite group, the number of elements in any subgroup H, denoted by $|H|$, is a divisor of $|G|$.*

Proof: Consider the *cosets* of G of the form $g\mathrm{H} = \{ gh \mid h \in \mathrm{H} \}$, where g is some element of G. Using the group axioms, it can be shown that these cosets form a partition of the elements of G: note $g_1\mathrm{H}$ and $g_2\mathrm{H}$ are identical sets if and only if $g_2^{-1}g_1 \in \mathrm{H}$, and each element of G belongs to exactly one distinct coset. Because all elements of the group can be partitioned into sets of size |H|, |G| must be a multiple of |H|. ∎

A *ring* R has two associated operators, normally termed + and × (or addition and multiplication, respectively), in which $(\mathrm{R}, +, 0)$ forms an Abelian group, and multiplication is closed and associative with respect to the set R. The additional ring axiom provides a link between the two operations:

5. Distributive: $\forall\, a, b, c \in \mathrm{R}, \quad a \times (b + c) = (a \times b) + (a \times c)$.

The additive identity is usually denoted by 0, and the additive inverse of a is written $-a$. If multiplication has an identity element, it is called 1, and the ring is said to have a unit element; if multiplication is commutative, the ring is said to be commutative. We will always deal with commutative rings with a unit element. Such a ring will be denoted by $(\mathrm{R}, +, \times, 0, 1)$. Some examples of rings are: the integers, $(\mathrm{Z}, +, \times, 0, 1)$, where addition and multiplication are the familiar integer operations; the integers modulo m, where m is an integer, $(\mathrm{Z}, + \pmod m, \times \pmod m, 0, 1)$; polynomials over the integers, written $(\mathrm{Z}[x], +, \times, 0, 1)$, where the operations are polynomial addition and multiplication; and polynomials over a field F modulo a polynomial $g(x)$, written $(F[x], + \pmod{g(x)}, \times \pmod{g(x)}, 0, 1)$. Typically, in writing out a product, the multiplication sign will be omitted; e.g., $a \times b = ab$.

2.2 Fields

A field F is a commutative ring in which the nonzero elements form a group under multiplication. In other words, every nonzero element a of the field has a multiplicative inverse, normally written a^{-1}. Although fields can be denoted similarly to rings, such notation is rarely used. Examples of fields include the rational numbers, the real numbers, the complex numbers, and the integers mod a prime p. From the above discussion, this latter structure forms a ring, and it is a straightforward task to prove that all elements of this ring have multiplicative inverses. Thus, the integers mod a prime p form a field which is denoted GF(p), for the Galois field with p elements. A subset of F which satisfies the field axioms is known as a subfield; for example, the rationals are a subfield of the reals, which are in turn a subfield of the complex numbers. One consequence of the field axioms is that addition, subtraction, multiplication, and division can be performed algebraically just as in the complex field. For example, methods such as the quadratic formula apply to all fields where the formula makes sense; similarly, as we shall see later, an n-point discrete Fourier transform can be computed in any field that contains a primitive n^{th} root of unity. Thus, to a

large extent, the algebraic methods learned in complex analysis can be employed over any field, finite fields in particular.

The reader will observe that some of the structures given in the above examples consist of a finite number of elements, while others have infinite size. For example, there are an infinite number of integers, but the integers mod m contain exactly m distinguishable elements. Although some properties apply only to finite structures (typically involving the size of some subset of elements, such as Lagrange's theorem), most of the results of abstract algebra apply equally to both finite and infinite structures. This generality allows us to understand many facets of finite fields by analogy with the more familiar complex arithmetic.

An important parameter of any field is known as the *characteristic*, which is informally defined as the number of times any nonzero element must be added to itself to produce zero. To show rigorously that this concept is well defined, consider the replication operator $\bullet$, which takes as arguments an integer n, known as the replication count, and a field element a, and returns a field element:

$$\begin{aligned} 0 \bullet a &= 0 \\ 1 \bullet a &= a \\ &\vdots \\ n \bullet a &= ((n-1) \bullet a) + a\,. \end{aligned}$$

Clearly any expressions in the replication count involve integer operations, while the expressions in a involve field operations. If, for $n < 0$, we define $n \bullet a = |n| \bullet (-a)$, then for all integers n,m, $(n \bullet a) + (m \bullet a) = (n+m) \bullet a$. Similarly, $n \bullet (m \bullet a) = (nm) \bullet a$. Now, for $n \neq 0$, and $a, b \neq 0$, it is clear that $n \bullet a = 0$ if and only if $n \bullet b = 0$, since

$$ba^{-1}(n \bullet a) = n \bullet b\,.$$

The characteristic is thus defined as the minimum n for which $n \bullet a = 0$ for any $a \neq 0$. If this number is infinite, as for example in the rationals or reals, the field is said to be of characteristic zero. The characteristic specifies the smallest subfield of the field. That is, clearly the multiplicative identity must be an element of any subfield, and by closure so must all elements of the form $x = n \bullet 1$, as well as the multiplicative inverse of x. For example, if the field is of characteristic zero, the smallest subfield is isomorphic to the rationals, since the elements $n \bullet 1$ constitute the integers, and the replication operator applied to the reciprocals of the integers produces the rest of the rationals. Obviously, any such field has an infinite number of elements, so no finite field can be of characteristic zero.

It can easily be shown that if a field does not have characteristic zero, it must have characteristic p, where p is prime. Suppose that the characteristic of a field F is $p = km$, where both k and m are greater than one. Then,

$$0 = p \bullet a = k \bullet (m \bullet a) = k \bullet b = ba^{-1}(k \bullet a)\,,$$

where $b = m \bullet a$, implying that $k \bullet a = 0$. However, p was the smallest such integer, and clearly $k < p$. So, by contradiction, p must be prime; in particular, any finite field must be of prime characteristic. It is not difficult to show, using the above properties of the operator $\bullet$, that the elements $n \bullet 1$, for $0 \leq n < p$, form a subfield isomorphic to GF(p), and because this is the smallest possible subfield of F, it is unique. Thus, we may without loss of generality refer to it as GF(p).

2.3 Extension Fields

From the above discussion, every field F contains a subfield which is isomorphic either to the rationals or to GF(p) for some p, depending on the characteristic of F. This fact can also be construed as a statement about the construction of fields. In other words, to build a large field, we must begin with the smallest field of the given characteristic. Consider the following theorem.

Theorem 2.2. *A finite field F of characteristic p is actually a vector space over GF(p).*

Proof: We will prove this theorem by constructing a basis for F. Clearly GF(p) is a subfield of F. Pick any nonzero field element as the first basis vector, and call it μ_0. Obviously, there are p distinct elements of the form $a\mu_0$, where $a \in$ GF(p), and $a\mu_0 = 0$ if and only if $a = 0$. To apply the induction step, suppose that we have a set of elements $B = \{ \mu_i \in F \mid i = 0, 1, \ldots, k-1 \}$, which are linearly independent with respect to GF(p); that is,

$$\sum_{i=0}^{k-1} a_i \mu_i = 0 \qquad \text{iff} \quad a_i = 0, \ i = 0, 1, \ldots, k-1 ,$$

where each a_i is an element of GF(p). Consider the subset K of F which is spanned by B. If K = F, the basis is complete. However, if K is a proper subset of F, pick μ_k to be any element of F which is not in K. Now observe that the new set $\{\mu_i\}$ is also linearly independent, since clearly $\mu_k \neq 0$, and if there exists a non-trivial linear combination of the μ_i's which is zero, then a_k must be nonzero, because the elements of the old set B are linearly independent. That is,

$$0 = \sum_{i=0}^{k} a_i \mu_i \quad \text{implies} \quad \mu_k = a_k^{-1} \sum_{i=0}^{k-1} a_i \mu_i = \sum_{i=0}^{k-1} (a_k^{-1} a_i) \mu_i ,$$

which is a contradiction because μ_k was chosen to be not in the span of the set B. This process continues until the set B spans the entire field. Now, because the set B is linearly independent with respect to GF(p), each element of F has a unique representation as a linear combination of the elements of the basis. ∎

Two results follow immediately from this theorem. First, observe that if there are m elements in the basis B, the field F consists of exactly p^m elements, so any finite field of characteristic p must have size p^m for some positive integer m. The field F is said to be an *extension field* of degree m over GF(p). The concept of extension fields carries over into infinite fields as well: m is defined to be the dimension of the vector space. Also, the underlying field does not have to be the smallest field available; e.g., complex numbers are a two-dimensional extension of the reals. Note that the degree of the extension does not have to be finite; for example, it is quite possible to have an infinite field of characteristic p, and both the reals and the complex numbers are infinite extensions of the rationals. However, for coding applications, finite extensions are sufficient.

Second, field addition can now be performed component-wise using GF(p) addition, regardless of the basis chosen, since

$$c = a + b = \sum_{i=0}^{m-1} (a_i + b_i) \mu_i ,$$

if $a = \sum_{i=0}^{m-1} a_i \mu_i$ and $b = \sum_{i=0}^{m-1} b_i \mu_i$. Thus, it is clear that any of the well-known methods of linear algebra can be applied here to produce a new basis which may be more suited to the particular task at hand. For purposes of implementation, we will choose $p = 2$; then field elements can be represented as vectors over GF(2), with each component a Boolean value, so that field operations lend themselves naturally to digital logic. For example, since GF(2) addition is equivalent to the Boolean exclusive-or (XOR) operation, a serial adder can always be implemented over $GF(2^m)$ using two shift registers and a single XOR gate. One particularly nice aspect of fields of characteristic two is that $1 + 1 = 0$ or $1 = -1$; i.e., because there is no distinction between addition and subtraction, all signs can be ignored.

Now the question remains: how do we construct finite extensions of a given field? A very familiar example will serve to illustrate the method. Let us choose as the underlying field the real numbers. The first step is to find an irreducible polynomial over the reals, such as $f(x) = x^2 + 1$; in this case we have $m = 2$. Obviously, there are many other such polynomials, but they are all of degree two and will produce the same extension field. Let i be a root of $f(x)$, so that $i^2 = -1$. We choose as our basis for the extension the first two integer powers of i, namely $\{1, i\}$. Clearly these two elements must be linearly independent with respect to the reals; otherwise, i would satisfy a linear polynomial over the reals, contradicting the irreducibility of $f(x)$. Each element z of the new field is represented by a pair of real values (x, y); in other words, $z = x + iy$. Addition over the field is performed component-wise:

$$z_1 + z_2 = (x_1 + iy_1) + (x_2 + iy_2) = (x_1 + x_2) + i(y_1 + y_2).$$

However, multiplication is somewhat more complicated, involving the fact that $i^2 = -1$:

$$z_1 z_2 = (x_1 + iy_1)(x_2 + iy_2) = x_1 x_2 + i(x_1 y_2 + x_2 y_1) + i^2 y_1 y_2 = (x_1 x_2 - y_1 y_2) + i(x_1 y_2 + x_2 y_1).$$

Similarly, the formula for division can be derived.

Of course, it is no surprise that we have just constructed the field of complex numbers. The importance lies in the method by which we obtained this new field. The fact that $f(x)$ was irreducible implies that the first m powers of a root, say α, are linearly independent with respect to the underlying field F. Alternatively, $f(x)$ is known as the *minimal polynomial* of α with respect to F; i.e., it is the polynomial of smallest degree over F which has α as a root. Essentially the elements of the extension field can be thought of as polynomials, with the root α of $f(x)$ being used as the indeterminate; in fact, the distinction between the root α and the indeterminate x begins to fade, since higher powers of α can be reduced using $f(x)$ to a polynomial in α of degree less than m. Given an irreducible polynomial $f(x)$ over F and a root α, the extension field denoted by $F[\alpha]$ is the smallest field containing both F and α; the extension is of degree m, where m is the degree of $f(x)$. Arithmetic in $F[\alpha]$ is defined to be polynomial arithmetic modulo $f(x)$, with coefficient arithmetic performed over F; in other words, using the fact that $f(\alpha) = 0$, powers of the form α^i for $i \geq m$ are reduced to polynomials in α of degree less than m.

Clearly the structure $F[\alpha]$ forms a ring; to prove that it is a field we need merely show that every nonzero element has a multiplicative inverse. Let $h(x)$ be a nonzero polynomial of degree less than m, and let $b = h(\alpha)$. Obviously, $b \neq 0$, since $f(x)$ is the minimal polynomial of α. Now, since $f(x)$ is irreducible, it is relatively prime to $h(x)$. Therefore, using Euclid's algorithm, we can find polynomials $u(x)$ and $v(x)$ such that

$$u(x)\,h(x) + v(x)\,f(x) = 1. \tag{2.1}$$

Evaluating (2.1) at $x = \alpha$, with $a = u(\alpha)$, we find $ab = 1$, since $f(\alpha) = 0$. In other words, $a = b^{-1}$. But by choosing $h(x)$ appropriately, we can generate any nonzero element b; in particular, for a given field element $b = \sum_{i=0}^{m-1} b_i \alpha^i$, pick $h(x) = \sum_{i=0}^{m-1} b_i x^i$. Again, note the lack of distinction between the indeterminate x and the root α. Now we have a constructive method for finding multiplicative inverses of nonzero elements of F[α], leading to the following theorem.

Theorem 2.3. *Given an irreducible polynomial $f(x)$ of degree m over a field F, where α is any root of $f(x)$, the set*

$$F[\alpha] = \left\{ \sum_{i=0}^{m-1} b_i \alpha^i \;\middle|\; b_i \in F \right\}$$

forms a field with respect to the operations of polynomial addition and multiplication mod $f(x)$.

Proof: Clearly the set F[α] forms a commutative ring, and we have just seen that all nonzero elements have multiplicative inverses. Therefore, F[α] is a field. ∎

Turning our attention strictly to finite fields, i.e., finite extension fields over GF(p), let us present a few well-known facts without proof. First, it can be shown that all finite fields must have the form given in Theorem 2.3 [29, section 5.5]. But for what m do irreducible polynomials of degree m exist over GF(p)? It turns out that there are irreducible polynomials of every degree for all p. In other words, a finite field of size q exists if and only if $q = p^m$ for some prime p and positive integer m. Further, any two finite fields of the same size can be proven to be isomorphic. Thus we can speak of the field with p^m elements without ambiguity; this field is known as GF(p^m). Armed with these facts, we can construct finite fields of any size using the method outlined above.

2.4 Primitive Elements

Theorem 2.1 provides a good understanding of the additive structure of F = GF(p^m) : field elements can be considered as vectors over GF(p), with addition performed component-wise. Although the representation of multiplication will vary depending on the particular basis chosen, we will show that the multiplicative group of F is quite simple. Let us denote the nonzero elements of the field F by F*; i.e., the multiplicative group of F. For any element $a \in$ F*, we define a^k as follows:

$$\begin{aligned} a^0 &= 1 \\ a^1 &= a \\ &\vdots \\ a^k &= a a^{k-1} \\ a^{-k} &= (a^{-1})^k \,. \end{aligned}$$

In other words, this exponentiation operator is to multiplication what $\bullet$ is to addition. With each element a of the group F* is associated a positive integer $|a|$, known as the *order* of a, which is defined as the minimum positive exponent k for which $a^k = 1$. In retrospect, the characteristic of the field F is just the additive order of the nonzero field elements; similarly, the concept of the order of an element can be defined for any group.

Consider now the set of elements $\langle a \rangle = \{ a^i \mid i = 0, 1, 2, \ldots \}$. Clearly $\langle a \rangle$ contains exactly $|a|$ distinct elements, and it is easy to see that $\langle a \rangle$ forms a subgroup of F* under multiplication;

in fact, $(\langle a\rangle, \times, 1)$ is isomorphic to $(Z, + \pmod{|a|}, 0)$. But, by Lagrange's theorem, $|a| = |\langle a\rangle|$ must be a divisor of $|\mathrm{F}^*|$. We have just proved the following corollary to Lagrange's theorem:

Corollary 2.4. *The order of every element of a group must be a divisor of the group size.*

The group $(\langle a\rangle, \times, 1)$ is known as *cyclic*, because all group elements can be expressed as some power of a; the element a is known as the *generator* of $\langle a\rangle$. Finite cyclic groups have a particularly simple structure, since, as mentioned above, they are isomorphic to the integers under addition mod the group size. We are now ready to prove that the multiplicative group of a finite field is cyclic.

Theorem 2.5. *The multiplicative group F^* of GF(p^m) is cyclic. In other words, there exists an element $\alpha \in F^*$ such that $|\alpha| = |F^*| = p^m - 1$. α is known as a* primitive element *of GF(p^m).*

Proof: Let $n = |\mathrm{F}^*| = p^m - 1$, and consider the polynomial $g(x) = x^n - 1$. By the previous corollary, every element of F^* is a root of $g(x)$, which therefore factors linearly over F:

$$g(x) = x^n - 1 = \prod_{a \in \mathrm{F}^*} (x - a).$$

Now, if $d|n$, $g_d(x) = x^d - 1$ divides $g(x)$, so $g_d(x)$ must also factor linearly. In other words, for every divisor d of n, there are exactly d elements of F^* which satisfy $g_d(x)$.

If we denote by N_d the number of elements of F^* of order d, clearly $N_d \neq 0$ only if $d|n$. Consider the Euler phi function $\phi(d)$, defined as the number of positive integers less than or equal to d which are relatively prime to d. For any integer d, by considering the prime factorization of d, it can easily be shown that

$$d = \sum_{r|d} \phi(r), \tag{2.2}$$

where the sum includes all positive factors of d, including 1 and d. We now claim that $N_d = \phi(d)$, which we shall show by induction on d. Clearly $N_1 = 1 = \phi(1)$. So, let us assume that for all proper divisors r of d, $N_r = \phi(r)$. From the above argument, there are exactly d elements of F^* which satisfy $g_d(x)$; clearly all elements of order d must come from among this set, as well as all elements of order $r < d$ where $r|d$. In other words,

$$N_d = d - \sum_{\substack{r|d \\ r<d}} N_r,$$

which can be simplified using (2.2) to find $N_d = \phi(d)$, as we were to show.

By induction, $N_n = \phi(n)$, so there are exactly $\phi(n)$ primitive elements of GF(p^m). ∎

The multiplicative structure of GF(p^m) can thus be characterized very simply. Observe that a primitive element of GF(p^m) is a primitive n^{th} root of unity, and the minimal polynomial of a primitive element is known as a *primitive polynomial.* Given a primitive element α, every nonzero field element $a = \alpha^i$ for some i, with $0 \leq i < n = p^m - 1$, so elements of F^* can be specified by their logarithm to the base α. In fact, one common method of implementing multiplication over finite fields is to use log and antilog tables. Also, by Theorem 2.5, primitive d^{th} roots of unity exist in GF(p^m) if and only if $d|n$; each such root can be expressed as $\alpha^{n/d}$ for some primitive element α.

2.5 Conjugation

The proof of Theorem 2.5 relies heavily on the interplay between field addition and multiplication in discussing the factorization of the polynomial $x^n - 1$. In fact, most of the algebraic results dealing with finite extension fields are intimately related to the roots of polynomials over the base field. A very familiar example is again found in the complex numbers. Given a root i of $x^2 + 1$ over the reals, observe that $-i$ is also a root. Thus, the linear mapping defined by $(x + iy)^* = x - iy$, known as complex conjugation, is an automorphism of the complex numbers which clearly does not alter the real numbers. In other words, the choice of i as the root was totally arbitrary; we could just as well have chosen $-i$, so interchanging these two roots cannot change the way field arithmetic works. In this particular case, it can be shown that complex conjugation is the only such (non-trivial) automorphism.

In general, *conjugation* is defined as a permutation of the roots of an irreducible polynomial $f(x)$ which produces an automorphism of the associated extension field, leaving the underlying field intact. Since arithmetic in the extension field is independent of the particular root chosen, all roots of $f(x)$ are conjugates. The set of such automorphisms forms a group which can be related to the structure of the extension field. Over finite fields, conjugation can be expressed very simply: the generator of the automorphism group of GF(p^m) with respect to GF(p) is the mapping $x \mapsto x^p$. Consider the following lemma:

Lemma 2.6. *In GF(p^m), $(x + y)^{p^k} = x^{p^k} + y^{p^k}$. In other words, raising to a power of p is a linear operation.*

Proof: For $k = 0$, the theorem trivially holds. For $k = 1$, using the binomial theorem, we have

$$(x + y)^p = \sum_{i=0}^{p} \binom{p}{i} \bullet (x^i y^{p-i}).$$

But, for $0 < i < p$, $\binom{p}{i}$ is divisible by p. Since the field is of characteristic p, all the intermediate terms drop out, and we are left with

$$(x + y)^p = x^p + y^p.$$

Now, assume by induction that the theorem holds for all $i < k + 1$. Then

$$(x + y)^{p^{k+1}} = \left((x + y)^{p^k}\right)^p = (x^{p^k} + y^{p^k})^p = (x^{p^k})^p + (y^{p^k})^p = x^{p^{k+1}} + y^{p^{k+1}}. \quad \blacksquare$$

In particular, for fields of characteristic two, squaring becomes a linear operation, as well as raising to any power of two. Using this lemma, it can easily be shown that the set of roots of an irreducible polynomial $f(x)$ over GF(p) is exactly the set of conjugates of a given root α, i.e., $\{\alpha^{p^i}\}$. Thus, all automorphisms of the field GF(p^m) which fix GF(p) have the form $x \mapsto x^{p^k}$ for some non-negative integer k. Observe that $k = 0$ produces the identity automorphism, but so does $k = m$, since every element of the field is a root of $x^{p^m} - x = (x - 0)(x^n - 1)$.

The subfields of F=GF(p^m) can also be determined by considering conjugation. Suppose that GF(p^d) is a subfield of F. Then, F^* must have a multiplicative subgroup of order $p^d - 1$, implying that $(p^d - 1)|(p^m - 1)$, which holds only if $d|m$. If $d|m$, then $x^{p^d} - x$ factors linearly with roots in F, and, in fact, the roots of this polynomial form the field GF(p^d). In other words, GF(p^m) has GF(p^d) as a subfield if and only if $d|m$. From Theorem 2.2, F forms a vector space of dimension m/d with respect to this subfield. Given a primitive element α of F with respect to the subfield, the roots of the primitive polynomial of α are given by the conjugates $\alpha^{p^{ik}}$. So conjugation is defined by both the underlying field and the extension field.

2.6 Linear Functions

The previous sections have provided a brief introduction into the arithmetic structure of Galois fields. In the ensuing sections, we examine various types of linear functions over finite fields of characteristic two; however, these results can be extended to all finite fields. Consider an arbitrary function $f(x)$ from GF(2^m) into itself. Given an ordering of the field elements x_i, $f(x)$ can be specified by 2^m values:

$$f(x_i) = a_i, \quad \text{for } i = 0, 1, \ldots, 2^m - 1.$$

Let φ_i be the Lagrangian interpolation polynomial of x_i with respect to these points; i.e.,

$$\varphi_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{2^m-1} \frac{x - x_j}{x_i - x_j}.$$

Clearly $\varphi_i(x_j) = \delta_{i,j}$, where δ is the Kronecker delta, and $\deg(\varphi) < 2^m$. Thus,

$$f(x) = \sum_{i=0}^{2^m-1} a_i \varphi_i(x) = \sum_{i=0}^{2^m-1} f_i x^i.$$

In other words, any function from GF(2^m) into itself can be expressed as a polynomial of degree less than 2^m.

Now suppose that $f(x)$ is a linear function; i.e.,

$$f(x + y) = f(x) + f(y), \quad \forall x, y \in \text{GF}(2^m).$$

Linearity implies that f can be entirely specified by its actions on a basis, since if $\{ \mu_i \mid i = 0, 1, \ldots, m-1 \}$ is a basis, then

$$f(x) = f\left(\sum_{i=0}^{m-1} x_i \mu_i \right) = \sum_{i=0}^{m-1} f(x_i \mu_i) = \sum_{i=0}^{m-1} x_i \bullet f(\mu_i),$$

where each $x_i \in$ GF(2). Every basis element can map into any field element, so there are exactly $(2^m)^m = 2^{m^2}$ distinct linear functions. By a simple counting argument, we can derive the polynomial form of $f(x)$. Because conjugation is a linear operation, the functions x^{2^i} are linear. Thus any function of the form $b(x) = \sum_{i=0}^{m-1} b_i x^{2^i}$ is linear, where each $b_i \in$ GF(2^m), and there are exactly 2^{m^2} such functions. Further, they are all distinct because $b(x)$ is identically zero if and only if each $b_i = 0$; otherwise a polynomial of degree 2^{m-1} would have 2^m roots. So, all linear functions from GF(2^m) into GF(2) must have the form

$$f(x) = \sum_{i=0}^{m-1} a_i x^{2^i},$$

where each $a_i \in$ GF(2^m). These functions may also be thought of as linear transformations, since GF(2^m) is an m-dimensional vector space over GF(2), so a linear function $f(x)$ can also be expressed as a binary matrix.

As a particular example which will prove useful later, let us examine the class of binary-valued linear functions over GF(2^m). Consider the linear function *trace*, denoted Tr(x), which is defined by

$$\mathrm{Tr}(x) = \sum_{i=0}^{m-1} x^{2^i}.$$

If $y = \mathrm{Tr}(x)$, then, applying the conjugation operator,

$$y^2 = \left(\sum_{i=0}^{m-1} x^{2^i}\right)^2 = \sum_{i=0}^{m-1} x^{2^{i+1}} = \mathrm{Tr}(x^2) = y,$$

since $x^{2^m} = x$. In other words, y is a root of the equation $0 = y^2 - y = y(y-1)$, implying that $y \in$ GF(2). Thus, the trace is a linear function from GF(2^m) into GF(2). How many such functions can there be? Given a basis $\{\mu_i\}$, each basis element can map into either 0 or 1, so there are exactly 2^m binary-valued linear functions. Now consider the class of linear transformations $b_\beta(x) = \mathrm{Tr}(\beta x)$, where $\beta \in$ GF(2^m). Clearly these functions are binary valued, and because $b_\beta(x)$ is a polynomial of degree 2^{m-1} or less, it is the zero function if and only if $\beta = 0$. Thus, there are 2^m distinct such functions. By this simple counting argument, we see that any binary-valued linear function must have the form $\mathrm{Tr}(\beta x)$ for some $\beta \in$ GF(2^m).

2.7 Discrete Fourier Transforms over Finite Fields

The concept of discrete Fourier transforms is well understood in the realm of complex numbers. However, an examination of the proofs for the various properties of the Fourier transform reveals only two essential requirements: arithmetic must take place over a field, and there must be a primitive n^{th} root of unity in the field. From the preceding discussions it is thus clear that Fourier transforms can be defined over finite fields as well, with a single restriction. In the complex case, a primitive root of unity is available for every integer n and is given by $\alpha = e^{-i\frac{2\pi}{n}}$, but over GF($2^m$), primitive n^{th} roots of unity exist only when $n|(2^m - 1)$. Given this constraint we make the following definition.

Definition. Consider the field GF(2^m), and a positive integer n which divides $2^m - 1$. Given a vector of field elements $\mathbf{v} = (v_0, v_1, \ldots, v_{n-1})$, and a field element α with $|\alpha| = n$, the *Fourier transform* of $\mathbf{v}$, denoted by $\mathbf{V} = (V_0, V_1, \ldots, V_{n-1})$, is defined as

$$V_k = \sum_{i=0}^{n-1} v_i \alpha^{ik}. \tag{2.3}$$

By analogy with the complex case, we may call the index i *time* and the index k *frequency*, although there is no physical significance associated with these terms. Similarly, the vector $\mathbf{v}$ is known as the *time-domain* vector, while the vector $\mathbf{V}$ is called the *frequency-domain* vector or *spectrum*. It will be our convention for the rest of this thesis that, where the distinction makes sense, lower-case vectors will represent time-domain signals, while upper-case vectors are the corresponding frequency-domain vectors. For example, components of received (and possible garbled) Reed-Solomon words will be denoted by s_i, while the syndromes, which form part of the Fourier transform of $\mathbf{s}$, will be written S_k.

We now proceed to state (without proof) many of the properties of the Fourier transform as defined above; most of these properties are identical to those of the complex transform. Again, these facts can be generalized to fields of characteristic $p \neq 2$, but several simplifications result from considering only GF(2^m). The interested reader should refer to Blahut [10, chapter 8] for an detailed explanation of these concepts, complete with proofs, since these theorems are taken directly from Blahut's presentation, adapted to the case $q = 2^m$.

Theorem 2.7. *(Inversion) A vector is related to its Fourier transform, as defined in (2.3), by*

$$v_i = \sum_{i=0}^{n-1} V_k \alpha^{-ik}. \quad \blacksquare$$

Theorem 2.8. *(Convolution) If $e_i = f_i g_i$ for $i = 0, 1, \ldots n-1$, then*

$$E_j = \sum_{k=0}^{n-1} F_{((j-k))} G_k, \qquad \text{for} \quad j = 0, 1, \ldots, n-1,$$

where the double parentheses represent modulo n arithmetic on the indices. $\blacksquare$

Theorem 2.9. *(Translation) If $\{ v_i \} \Leftrightarrow \{ V_k \}$ is a Fourier transform pair, then the following are also Fourier transform pairs:*

$$\{ \alpha^i v_i \} \Leftrightarrow \{ V_{((k+1))} \}$$
$$\text{and} \qquad \{ v_{((i-1))} \} \Leftrightarrow \{ \alpha^k V_k \}. \quad \blacksquare$$

Theorem 2.10. *(Polynomial Representation) If we represent the Fourier vectors* $\mathbf{v}$ *and* $\mathbf{V}$ *as polynomials,*

$$v(x) = \sum_{i=0}^{n-1} v_i x^i \qquad \text{and} \qquad V(x) = \sum_{k=0}^{n-1} V_k x^k,$$

then

$$V_k = v(\alpha^k) \qquad \text{and} \qquad v_i = V(\alpha^{-i}). \quad \blacksquare$$

This last theorem makes clear a certain duality between polynomials and Fourier vectors, which we shall often use implicitly. At certain points during a discussion it will be convenient to refer to vectors as polynomials, and at other times we will find it more useful to refer to them as vectors. Often the transition between points of view will be quite abrupt, depending on the need at hand, so the reader should familiarize himself with these concepts. The Fourier transform will allow us to couch many coding theory concepts in terms which are already familiar from traditional complex analysis.

Chapter 3

Coding Theory

3.1 Linear Block Codes

For any digital communication channel, there is a discrete set of values, sometimes termed *letters* or *characters*, which can be sent and received; often, the letters will be individual bits. A particular sequence of such letters is known as a *word*. For our application, a *code* is defined as a set of words known as *codewords* which can be transmitted over a channel. Typically the information to be sent is divided into packets, and each packet is *encoded* by some rule to produce a codeword before being modulated and sent. The receiver takes the incoming (and possibly garbled) word from the demodulator and *decodes* it to produce the most likely candidate for the original codeword, from which the original information is then extracted, as shown in Figure 3-1. Hopefully, the introduction of coding provides some system advantage, because clearly there is a cost associated with building the encoder and decoder. This *coding gain* is often measured in reliability or in the use of a simpler (less expensive) modulator/demodulator on the given channel.

In this chapter we will examine the basic concepts of linear block codes in general and Reed-Solomon codes in particular. In a *block code*, each codeword is composed of exactly n characters: n is known as the *block length* of the code. By contrast, in another major class of codes, known as *convolutional codes*, the codewords contain an infinite stream of characters; in practice the sequence is truncated at some point. Although convolutional codes have many important applications, we will not consider them in detail in this thesis. For digital implementation, each letter of the channel alphabet can be represented by a fixed number of bits, so we may consider a character to be an element of a finite field F; thus, codewords in a block code form vectors of length n over F:

$$\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}),$$

where each $c_i \in \mathrm{F}$. A code is said to be *systematic* if the information characters always appear unaltered in some subset of the letter positions; the remaining locations in the codeword are known as *parity* or *check* locations. Systematic codes can be desirable because, if an uncorrectable error pattern occurs, often a large portion of the information characters are nonetheless correct, so much of the data is recoverable. *Linear block codes* have the property that the sum of any two codewords is also a codeword, where addition is performed componentwise. In other words, linear

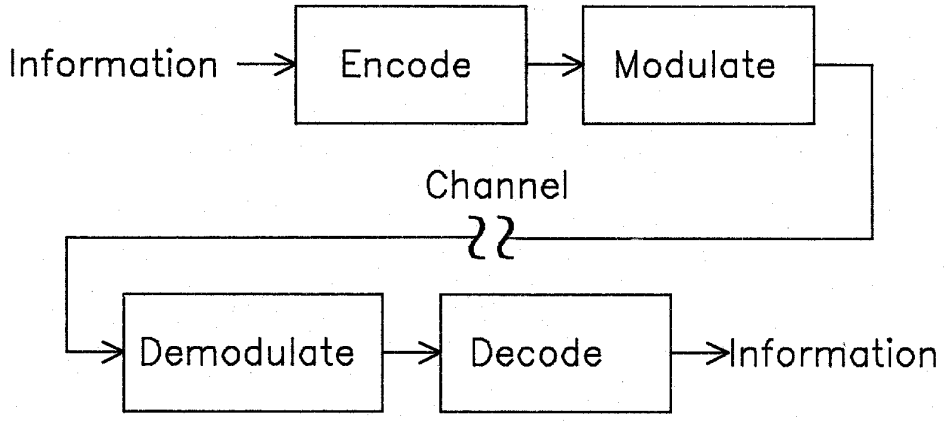

Figure 3-1. Coding on a Digital Channel

block codes form a subspace, say of dimension k, of a vector space of dimension n over F. Such codes are fairly well understood because they lend themselves to analysis using standard methods of linear algebra. For example, it can be shown that, by using an appropriate encoding technique, every linear block code is systematic. If the dimension of a linear code is k, the corresponding code is known as an (n, k) code, meaning that there are k information letters and $n - k$ parity characters; the code is said to have *redundancy* $n - k$.

For a given application, the complexity of an appropriate code and its associated encoding and decoding rules will depend on both the error statistics of the given channel and the desired level of reliability. For example, if the raw error probability is acceptable, then the information can be sent unaltered over the channel. However, if higher reliability is required, redundant information must be added to allow the decoder to reconstruct the transmitted codeword and thus extract the desired information. Let us define the *rate* of a code to be the ratio of the number of information bits to the total number of bits in the codeword. The reader is cautioned that, according to this definition, rate has nothing to do with speed; instead, it is a measure of the efficiency of the code: the lower the rate, the more overhead is required, in terms of redundant information. Obviously, the rate R satisfies the inequality $0 \leq R \leq 1$, and an (n, k) linear code has rate $R = k/n$. One simple coding scheme is known as the $n = 3$ repetition code: the encoder sends the information three times and the decoder takes a majority vote on the three received messages. Such codes exist for all positive n and have rate $1/n$; in fact, adding no redundancy corresponds to the $n = 1$ repetition code. Although repetition codes allow simple implementation, they are not very powerful; usually, codes with much higher rates can be used to achieve the same level of reliability, at the cost of more complex encoders and decoders.

Stated formally, Shannon's channel coding theorem [46] asserts that for any discrete memoryless channel there exists a number C, known as the *channel capacity*, with $0 \leq C \leq 1$, such that for all $\epsilon > 0$ and all $R < C$, there exists a code with rate $\geq R$ and an accompanying decoding algorithm which, when used on the given channel, has a probability of decoding error which is less than ϵ. Of course, no actual channel is perfectly discrete and memoryless, but the theorem has been extended to several other channel models [40], and in many instances the channel of interest approximates one of these models fairly closely. Shannon's theorem provides

an existence proof for good codes, as the block length of the code becomes asymptotically large; however, the channel coding theorem gives no hints about the construction of such codes. Further, in real life we must contend with finite block length, so it is not clear initially whether Shannon's result can be meaningfully applied to practical problems. Fortunately, many powerful linear block codes have been discovered which are quite useful in practice, although they may not meet Shannon's asymptotic bound.

3.2 Distance Metrics

A grasp of the distance metric used in coding theory is fundamental to an understanding of how codes can be used to correct errors. The *Hamming weight* of a vector $\mathbf{x} = (x_0, \ldots, x_{n-1})$, denoted $w_H(\mathbf{x})$, is defined as the number of nonzero components of $\mathbf{x}$. If we define a scalar function

$$\delta(x) = \begin{cases} 0, & x = 0; \\ 1, & x \neq 0, \end{cases}$$

then the Hamming weight can be formally defined by

$$w_H(\mathbf{x}) = \sum_{i=0}^{n-1} \delta(x_i).$$

Let us define the *Hamming distance* between two vectors $\mathbf{x}$ and $\mathbf{y}$, denoted $d_H(\mathbf{x}, \mathbf{y})$, as the number of components in which the two vectors differ. Clearly, the Hamming distance is just the Hamming weight of the difference between the two vectors; i.e.,

$$d_H(\mathbf{x}, \mathbf{y}) = w_H(\mathbf{x} - \mathbf{y}) = \sum_{i=0}^{n-1} \delta(x_i - y_i).$$

Further, it can be easily shown that the Hamming distance obeys the triangle inequality, so it is a metric in the rigorous sense of the term over any given vector space.

An important parameter of any code C is known as the *minimum distance* or $d_{\min}$, which is defined as the minimum Hamming distance between any two distinct codewords:

$$d_{\min}(C) = \min_{\substack{\mathbf{x},\mathbf{y} \in C \\ \mathbf{x} \neq \mathbf{y}}} d_H(\mathbf{x}, \mathbf{y}) = \min_{\substack{\mathbf{x},\mathbf{y} \in C \\ \mathbf{x} \neq \mathbf{y}}} w_H(\mathbf{x} - \mathbf{y}).$$

Now suppose that $\mathbf{x}$ and $\mathbf{y}$ are two distinct codewords of a linear block code. Then the vector $\mathbf{z} = \mathbf{x} - \mathbf{y}$ is also a codeword; choosing $\mathbf{y}$ to be the zero vector, we clearly see that all nonzero codewords can be expressed in this form. So, if C is a linear block code, then the expression for the minimum distance can be simplified to

$$d_{\min} = \min_{\substack{\mathbf{x} \in C \\ \mathbf{x} \neq \mathbf{0}}} w_H(\mathbf{x}).$$

In other words, to find $d_{\min}$ it is sufficient to find a nonzero codeword with the lowest possible weight. Conceptually, a minimal weight codeword can be found by exhaustive search, although such a search is practically feasible for only the simplest codes. Instead, an algebraic construction is typically given for the codewords, allowing the minimum weight to be determined analytically.

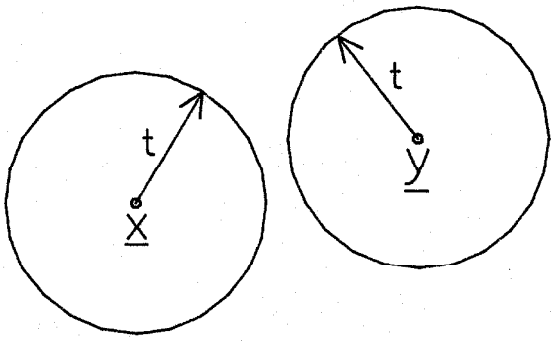

Figure 3-2. Using Hamming Distance to Correct t Errors

The parameter $d_{\min}$ determines how many errors can be corrected by a code. Suppose that a codeword $\mathbf{x}$ is transmitted over the channel, and a possibly corrupted version $\mathbf{x}+\mathbf{y}$ is received. The number of character errors which have occurred is obviously $e = w_H(\mathbf{y})$. For what values of e can a decoder correctly determine the original codeword x or, equivalently, the error pattern $\mathbf{y}$? The answer to this question is given in the following theorem, which we will prove by geometric arguments.

Theorem 3.1. *If a code has minimum distance $d_{\min} = 2t+1$, then any error pattern of weight t or less can be corrected.*

Proof: We will prove this result by giving a decoding algorithm. With the Hamming distance metric, it is possible to define a sphere of radius r about a vector $\mathbf{x}$ as the set of vectors which lie within Hamming distance r of $\mathbf{x}$. Consider spheres of radius t about each codeword in the vector space. Because $d_{\min} = 2t+1$, all of these spheres are disjoint, as shown in Figure 3-2. If a codeword $\mathbf{x}$ is transmitted, then the received vector $\mathbf{x}+\mathbf{y}$ lies within the sphere centered about $\mathbf{x}$ if and only if $e = w_H(\mathbf{y}) \leq t$. Therefore, the error pattern can be decoded (i.e., corrected) by the following algorithm: pick the codeword which is closest in Hamming distance to the received vector. By the above argument, this procedure is well-defined and guaranteed to produce a unique result as long as $e \leq t$. ∎

Notice that the theorem provides only a sufficient condition; in other words, if $d_{\min} = 2t+1$ it may be possible to correct some error patterns of weight greater than t. Unfortunately, the algorithm presented in the above proof is rarely feasible to apply in practice, but it does give an indication of the goal of a *maximum-likelihood* decoder. Basically, the idea is to produce a codeword which is closest to the received vector by an appropriate metric, so that the output of the decoder is the most likely transmitted codeword. If errors in distinct characters are roughly independent, then the Hamming distance is probably an acceptable metric, while if errors tend to occur in bursts (or more complicated patterns), a different metric will be necessary. For many applications the Hamming metric is used to design a code, and then variations such as interleaving (which will be discussed later) are employed to match the characteristics of the code to the channel at hand.

Observe that, at least conceptually, a maximum likelihood decoder performs some sort of correlation of the received word with every codeword. Thus far we have discussed errors as a binary phenomenon; however, often the demodulator can extract analog information which can be

used to assign probabilities that the given received value actually corresponds to each letter of the channel. For example, suppose that a value of ± 1.0 is transmitted to convey binary information, but the receiver can observe any real value. If we assume that the channel introduces noise with a Gaussian distribution, then upon reception of the value 0.9 we have some level of confidence that a $+1.0$ was in fact sent. Taking such information into account in the decoding process is often termed soft decision decoding, as opposed to hard decision decoding, in which all input values are quantized to the nearest possible channel letter before the decoding process begins. On most communication channels, utilizing soft information can greatly enhance the error-correcting capability of a code, at the cost of additional complexity in the decoder. Convolution codes are attractive because, using the Viterbi algorithm [40], considerable soft information can be incorporated into the decoder with very little overhead.

In general, it is much more difficult to incorporate soft information into the decoding process for block codes, although some advances have been made in this direction [6]. However, one very simple type of soft decision can be readily utilized in Reed-Solomon and related codes. Often, the demodulator can determine that the letter currently being received is in error; such a character is termed an *erasure*. If there are r channel letters, an erasure effectively assigns a probability of $1/r$ to each of the possible characters. Although this information is fairly weak, as compared to a more complex comparison and probability assignment, introducing erasures into a decoder can produce significant gains in reliability for a given channel. Each erasure effectively reduces the distance of the code by one, since a decoder can no longer determine which codeword(s) differ from the received word in that component. Thus, by Theorem 3.1, each pair of erasures decreases the number of correctable errors by one; equivalently, each erasure corresponds to half an error. Later in the chapter we will discuss more powerful schemes, known as concatenated codes, which can take advantage of the best features of block codes without sacrificing the coding gain provided by soft decision decoding.

3.3 BCH and Reed-Solomon Codes

From a pedagogical point of view, it is perhaps easiest to define and explain the properties of Reed-Solomon codes in terms of the Fourier transform over finite fields [10]. As we shall see below, such a formulation is quite similar to the original exposition by Reed and Solomon [45], although for implementation purposes a more algebraic approach has often been adopted. Using the properties of the DFT outlined in the previous chapter, all of these viewpoints can be related fairly simply, and it is also possible to derive the more general class of BCH codes, which are named after their discoverers, Bose, Chaudhuri, and Hocquenghem. Using Fourier transforms to explain linear block codes has the added benefit of allowing engineers to utilize intuition which they have already developed about the DFT. In fact, in retrospect it has become clear that much of coding theory is closely connected with standard techniques of digital signal processing.

Definition. Consider the field GF(2^m), and a positive integer n which divides $2^m - 1$. Let α be a primitive n^{th} root of unity in GF(2^m). Given an integer L, where $0 \leq L < n$, an $(n, n-r)$ *Reed-Solomon code* over GF(2^m) is defined to be the set of all vectors

$$\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$$

of length n over the field whose Fourier spectrum with respect to α,

$$\mathbf{C} = (C_0, C_1, \ldots, C_{n-1}),$$

where $C_k = \sum_{i=0}^{n-1} c_i \alpha^{ik}$, satisfies $C_{((L+k))} = 0$ for $k = 0, 1, \ldots, r-1$. ∎

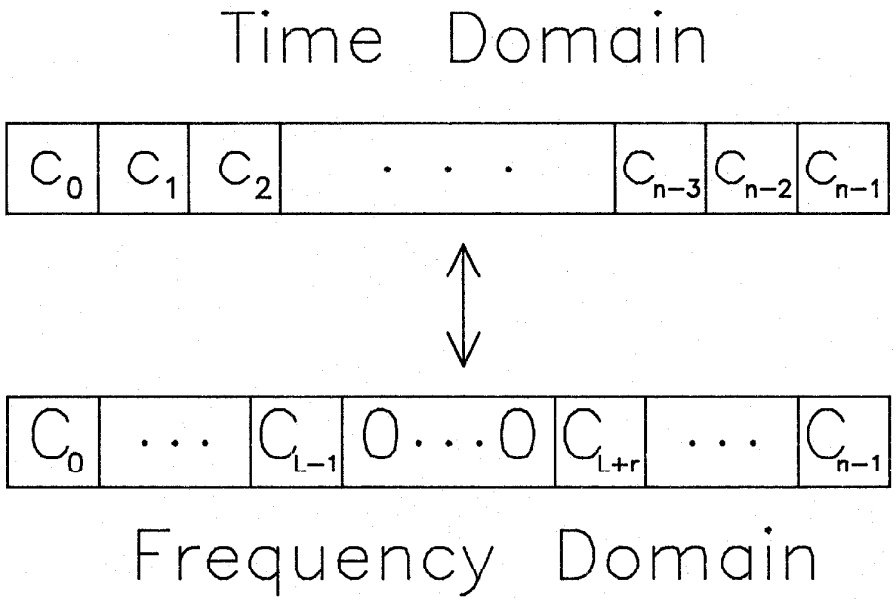

Figure 3-3. Frequency Constraints on Reed-Solomon Codewords

In other words, an RS code is defined to be the set of all vectors whose spectral components are zero in a fixed frequency window, as illustrated in Figure 3-3. Clearly the $(n, n-r)$ code has redundancy r, and there are $n-r$ information characters per codeword. The figure suggests that one encoding method is to start in the frequency domain with any spectrum which is zero in the specified window and then to perform an inverse transform, producing a valid time-domain codeword. Unfortunately, such an encoding rule is not systematic, and encoder implementations based around the inverse transform are not particularly efficient. There are two other equivalent interpretations of Figure 3-3. First, using the polynomial representation of $\mathbf{c}$ (Theorem 2.10), observe that the definition implies

$$c(\alpha^{L+k}) = 0$$

for $k = 0, 1, \ldots, r-1$. In other words, $c(x)$ is divisible by the *generator polynomial* of the code,

$$g(x) = \prod_{k=L}^{L+r-1} (x - \alpha^k),$$

which has degree r. A systematic encoding rule using this viewpoint is to interpret the information characters p_i as a polynomial

$$p(x) = \sum_{i=0}^{n-r-1} p_i x^i,$$

and let $h(x) \equiv x^r p(x) \pmod{g(x)}$ be the remainder when $x^r p(x)$ is divided by $g(x)$. The polynomial $c(x) = x^r p(x) - h(x)$ is then clearly divisible by $g(x)$ and is thus a codeword. Very simple encoders

can be built to perform this polynomial division [5]. In the second alternative viewpoint, which is essentially identical to the original presentation of Reed and Solomon, the frequency-domain vector is viewed as polynomial:

$$C(x) = \sum_{k=0}^{n-1} C_k x^k = x^{L+r} \sum_{k=0}^{n-r-1} C_{L+r+k} x^k = x^{L+r} D(x),$$

where $\deg(D) < n-r$, since $x^n = 1$ for all values of the field. Then

$$c_i = C(\alpha^{-i}) = \alpha^{-i(L+r)} D(\alpha^{-i}). \tag{3.1}$$

Although this interpretation is not particularly attractive for implementation, the minimum distance properties of the code can be easily derived from (3.1):

Theorem 3.2. *The minimum distance of an $(n, n-r)$ Reed-Solomon code is $d_{\min} = r+1$.*

Proof: Suppose instead that there is a nonzero codeword of weight $e < r+1$. Then $c_i = 0$ for all but e indices; in other words, by (3.1), the polynomial $D(x)$ has $n-e > n-r-1$ roots. However, no polynomial can have more roots over a field than its degree, and we saw above that $\deg(D) \leq n-r-1$. So, $d_{\min} \geq r+1$. Now considering **c** as a polynomial, observe that $c(x) = g(x)$ is a codeword and has weight $r+1$, so $d_{\min} \leq r+1$. ∎

Each redundant character thus increases the distance of a Reed-Solomon code by exactly one; clearly the addition of a check character can do no more than this, and such codes are known as *maximum distance separable*. Typically we will choose $r = 2t$ in order to be able to correct t errors. One nice feature of Reed-Solomon codes is the ease with which code rate can be traded for correction capability: two extra check characters always allow one additional error (or two erasures) to be corrected. While it is in theory possible to correct $2t$ erasures, this course is rarely chosen, because there is insufficient redundancy to help insure that additional errors have not occurred. Also observe that, when inspected on a bit level, RS codes have an inherent capability of correcting error *bursts*, since a single code error can correspond to up to m consecutive bit errors. In particular, a t error-correcting code over $\mathrm{GF}(2^m)$ can always handle error bursts up to $1+m(t-1)$ bits long, as well as some slightly longer bursts, depending on how the bit errors are aligned with respect to character boundaries. Further, for $t > 1$, *multiple* smaller bursts can be corrected. Thus, RS codes are well-suited to a mixture of random and burst errors.

Both Reed-Solomon and BCH codes are called *cyclic* because any circular rotation of the components of a codeword produces another codeword. To prove this assertion, note that, given a codeword $c(x)$, the polynomial $xc(x)$ is formally a codeword also, since it has zeroes at all the appropriate locations; however, it has degree n:

$$xc(x) = \sum_{i=0}^{n-1} c_i x^{i+1} = c_{n-1} x^n + \sum_{i=1}^{n-1} c_{i-1} x^i.$$

But since $g(x)$ is a divisor of $x^n - 1$, $x^n \equiv 1 \pmod{g(x)}$. That is,

$$xc(x) \equiv c_{n-1} + \sum_{i=1}^{n-1} c_{i-1} x^i \equiv \sum_{i=0}^{n-1} c_{((i-1))} x^i \pmod{g(x)},$$

where the double parentheses again indicate arithmetic mod n. The latter sum is exactly a circular rotation of $c(x)$, and since it is congruent to $xc(x)$ mod the generator polynomial $g(x)$, it must also be a codeword. Repeated applications of this argument prove that all rotations of $c(x)$ are in fact codewords. An alternative proof of this fact relies on the translation properties of the Fourier transform: a circular shift in the time domain corresponds to multiplying each frequency component C_k by α^k, which does not change the zero pattern, thus assuring that the rotated vector is also a codeword. This property will prove quite useful for implementing portions of the decoder in hardware.

If greater burst protection is required, a technique known as *interleaving* can be employed to enhance the burst-correction capabilities of the code. Interleaving involves shuffling characters from different codewords so that bursts are distributed evenly throughout the codewords. For example, consider a technique known as block interleaving, to depth $d = 4$; in other words, four codewords **a**, **b**, **c**, and **d** are encoded by row and transmitted by column:

$$a_0, b_0, c_0, d_0, a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2, \ldots, a_{n-1}, b_{n-1}, c_{n-1}, d_{n-1}\,.$$

More sophisticated schemes are possible [22], but for our purposes it is sufficient to know that interleaving to depth d roughly improves the burst error correction capability of a code by a factor of d. In general, interleaving is used when the additional redundancy needed to correct a burst directly is either unavailable (because of block length limitations) or too costly to implement.

At first, the restriction of block length to a divisor of $2^m - 1$ could seem a significant limitation. In some sense this is correct, but only in the case where a block length longer than $2^m - 1$ is desired. Given a code of natural block length n, it is always possible to shorten the code to any length n_0, where $n > n_0 > r$. The systematic encoding mentioned above which involves the generator polynomial $g(x)$ suggests the construction for such a *shortened code*. If both the encoder and decoder assume that all the high-order information characters (i.e., p_i for $i > n_0 - r - 1$) are zero, then these characters never have to be transmitted. Also, for the most part it turns out that the only major effect on the encoder and decoder is that cycles of length n are replaced by cycles of length n_0. Thus, up to the limit of the field size, it is possible to select both the redundancy and the block length as desired. It should however be noted that shortened codes are no longer cyclic. One goal of our decoder structures will be to handle shortened codes in an efficient manner.

Many other important codes can be derived using similar arguments. For example, it is possible to append up to two information symbols to a Reed-Solomon codeword of natural length n [10]; however, these *extended* Reed-Solomon codes are not of particular interest here. Also, it should be noted that the previous theorem applies to any code which has a zero frequency window of length r. Suppose that we wish to design a code over a subfield of GF(2^m); e.g., a binary code. Then all coefficients of the generator polynomial must be elements of the subfield as well, so $g(x)$ must have as roots not only α^i for $i = L, L+1, \ldots, L+r-1$ but all conjugates of these elements (with respect to the subfield) as well. Such codes are known as the *generalized BCH codes*, of which the binary BCH codes [12, 30] are the best known example. In terms of the spectrum, the codewords must have zero components in the frequency window of interest and in other spectral locations determined by the conjugacy constraints. The degree of the generator polynomial and thus the amount of redundancy for a code with given $d_{\min}$ will therefore increase as the subfield decreases in size. Reed-Solomon codes are a special instance of generalized BCH codes in which the two fields are identical; thus they are the only codes in the family which are

maximum distance separable. In particular, each binary BCH code is a subfield subcode of a Reed-Solomon code, so the decoder structures we will discuss in chapter six can handle binary BCH codes as well, although perhaps somewhat inefficiently.

3.4 The Key Equation

Given the definition of Reed-Solomon codes, it is a fairly straightforward task to design an encoder, at least conceptually. However, the arguments in the preceding section give no clue about possible decoder algorithms or implementations. Subsequent chapters will deal with this subject in depth. In preparation for such discussions, it will be instructive to derive a formal statement of the decoding problem, using the Fourier transform. An alternative proof of the minimum distance property of Reed-Solomon codes will serve as motivation for our derivation of what has come to be known as the *key equation*, the equation which must be solved by a decoder. It will be clear that this key equation also holds for the entire family of generalized BCH codes.

Our second proof of Theorem 3.2 proceeds as follows. Suppose that an $(n, n-r)$ RS code has minimum distance less than $r+1$; in other words there exists some nonzero codeword $\mathbf{c}$ with $w_H(\mathbf{c}) = e < r+1$. Let I be the set of indices for which c_i is non zero:

$$\mathrm{I} = \{\, i \mid c_i \neq 0 \,\}.$$

Clearly I contains exactly e distinct elements. Alternatively, indexing the components of $\mathbf{c}$ by field elements instead of integers, we may make the association $c_i \mapsto c_{\alpha^i}$; then I corresponds to a set of field elements. We will often talk about these two representations interchangeably.

Consider now the polynomial

$$\Lambda(x) = \prod_{i \in \mathrm{I}} (\alpha^i x - 1) = \sum_{j=0}^{e} \Lambda_j x^j .$$

Note that the roots of $\Lambda(x)$ correspond to the nonzero components of $\mathbf{c}$; in other words,

$$\Lambda(\alpha^{-i}) = 0 \qquad \text{if and only if} \qquad c_i \neq 0 . \tag{3.2}$$

Using the polynomial representation property of the Fourier transform, we may assume that $\Lambda(x)$ corresponds to a frequency-domain vector. If we let $\lambda(x)$ be the associated time-domain vector, then observe that (3.2) implies

$$\lambda_i c_i = 0 \quad \text{for} \quad i = 0, 1, \ldots, n-1 . \tag{3.3}$$

So, by Theorem 2.8 and (3.3), the convolution of $\Lambda(x)$ and $C(x)$ in the frequency domain is also zero. Since Λ is of degree e, we find that, for all k,

$$0 = \sum_{j=0}^{e} \Lambda_j C_{((k-j))} ,$$

or, since $\Lambda_0 = 1$,

$$C_k = -\sum_{j=1}^{e} \Lambda_j C_{((k-j))} . \tag{3.4}$$

In other words, the components of the Fourier transform of **c** satisfy a (circular) linear recurrence of length $e < r + 1$. However, since **c** is a codeword, its spectrum is zero in a window of size r. Beginning with this window, and applying (3.4) to complete the spectrum **C**, we see that in fact all components are zero. Thus, **c** is the zero vector as well, so by contradiction all nonzero codewords have weight greater than r.

But now we can interpret (3.4) as a statement of the decoding problem. Let us assume $r = 2t$ for simplicity, although such an assumption is not necessary. Given a received vector $s(x) = c(x) + e(x)$, where $c(x)$ is a codeword and $e(x)$ is an error pattern of weight $e < t + 1$, our goal is to find $c(x)$, or equivalently, $e(x)$. Let $\sigma(x)$ be the polynomial whose roots correspond to the nonzero coefficients of $e(x)$, similar to $\Lambda(x)$ above. In this instance, $\sigma(x)$ is known as the *error-locator* polynomial, because its roots determine the location of the errors in $e(x)$. If we let **E** be the Fourier transform of the error pattern, then the argument leading up to (3.4) can also be applied in this case to yield:

$$E_k = -\sum_{j=1}^{e} \sigma_j E_{((k-j))} . \tag{3.5}$$

But now consider the frequency window in which codewords have zero spectral components. Here, $C_k = 0$, implying $S_k = E_k$, so we can compute a window into the spectrum of the error pattern directly from the received word. Then, for $k = L, L+1, \ldots, L+2t-1$,

$$S_k = -\sum_{j=1}^{e} \sigma_j S_{((k-j))} . \tag{3.6}$$

The values S_k for $k = L, L+1, \ldots, L+2t-1$ are known as the *power-sum syndromes*. If we can determine the coefficients of the recurrence relation (3.6), then the entire error spectrum can be determined recursively from the syndromes, allowing the error pattern to be computed by an inverse transform. This view of the problem is often known as *transform decoding*, in which the syndromes are considered part of a Fourier transform and the error pattern is obtained from an inverse transform of the completed error spectrum. In fact, any decoding method which utilizes the power-sum syndromes can be considered a transform decoder, although the method of computing the error pattern given $\sigma(x)$ may differ significantly from the procedure described above.

Suppose now that we define a polynomial

$$S(x) = \sum_{k=0}^{2t-1} S_{L+k} x^k .$$

Let I be the set of indices corresponding to nonzero components of **e**, and consider the product $S(x)\sigma(x)$. First we note that, in the window of interest, the syndromes are related to the error pattern by

$$S_k = s(\alpha^k) = e(\alpha^k) = \sum_{i \in I} e_i \alpha^{ik} . \tag{3.7}$$

Using (3.7), we can derive the key equation:

$$\begin{aligned}
\sigma(x)S(x) &= \prod_{i\in I}(\alpha^i x - 1) \sum_{k=0}^{2t-1} S_{L+k}x^k \\
&= \prod_{i\in I}(\alpha^i x - 1) \sum_{k=0}^{2t-1}\Big(x^k \sum_{j\in I} e_j \alpha^{j(L+k)}\Big) \\
&= \prod_{i\in I}(\alpha^i x - 1) \sum_{j\in I}\Big(e_j\alpha^{jL} \sum_{k=0}^{2t-1} \alpha^{jk}x^k\Big) \\
&= \prod_{i\in I}(\alpha^i x - 1) \sum_{j\in I} e_j\alpha^{jL}\,\frac{\alpha^{2jt}x^{2t}-1}{\alpha^j x - 1} \\
&= \sum_{j\in I} e_j\alpha^{jL}(\alpha^{2jt}x^{2t}-1) \prod_{\substack{i\in I\\ i\neq j}}(\alpha^i x - 1) \\
&\equiv -\sum_{j\in I}\alpha^{jL}e_j \prod_{\substack{i\in I\\ i\neq j}}(\alpha^i x - 1) \pmod{x^{2t}} \\
&= \omega(x),
\end{aligned} \tag{3.8}$$

where $\omega(x)$, defined in the last line of (3.8), is known as the *error-evaluator* polynomial, for reasons which will become clear shortly. Note that $\deg(\omega) < e \leq t = \deg(\sigma)$. In other words, we need to look for two polynomials $\sigma(x)$ and $\omega(x)$, each of degree less than $t+1$, that satisfy

$$\sigma(x)S(x) \equiv \omega(x) \pmod{x^{2t}}. \tag{3.9}$$

This is the syndrome key equation. Algorithms to solve (3.9) will be presented in Chapter five.

Clearly the power-sum syndromes depend only on the error pattern, not on the codeword, while any other frequency component must depend on $c(x)$. In fact, the syndromes constitute a minimal set of information necessary to determine a correctable error pattern. The equivalent of the syndromes in the time domain is the remainder polynomial, $r(x) \equiv s(x) \pmod{g(x)}$, which can be easily calculated by reencoding the received word. Since $S_{L+k} = r(\alpha^{L+k})$ for all components in the window, it is possible to compute the syndromes given the remainder. Similarly, by the Chinese remainder theorem, the syndromes provide enough information to reconstruct $r(x)$. Decoding algorithms which utilize only $r(x)$ are known as remainder decoding techniques, and in Chapter five we shall see that there is a corresponding key equation for remainder methods.

3.5 Correcting the Errors

Solution methods for the key equation will be deferred until later, but given the error polynomials $\sigma(x)$ and $\omega(x)$, how does one go about producing the error pattern? The method outlined above, involving only the error-locator polynomial, is often referred to as *recursive extension* in the frequency domain. Because the error spectrum is known in the window, using (3.5) the entire spectrum $\mathbf{E}$ can be computed given $\sigma(x)$, and an inverse transform produces $\mathbf{e}$. While this approach is conceptually enlightening, in practice the computation is not particularly efficient in terms of area-time product.

For reasons to be discussed in the following section, we will be largely interested in high-rate codes, so a correction procedure involving only $\sigma(x)$ and $\omega(x)$, both of size less than the redundancy of the code, would be very attractive. Fortunately, a fairly simple algebraic technique can be used to produce the error values. Observe that s_k is in error if and only if $\sigma(\alpha^{-k}) = 0$, so consider the value of the error evaluator $\omega(x)$ at these points:

$$\omega(\alpha^{-k}) = \alpha^{kL} e_k \prod_{\substack{i \in I \\ i \neq k}} (\alpha^{i-k} - 1), \tag{3.10}$$

for $k \in I$. The product in (3.10) reminds us of the form of the derivative $\sigma'(\alpha^{-k})$, if the error-locator were defined over the complex field. In fact, a formal derivative can be defined for polynomials over any field, and this operator obeys all the differentiation rules of standard calculus. For example, $(uv)' = u'v + uv'$ and $(x^k)' = kx^{k-1}$, so over fields of characteristic 2, all terms of the form x^{2k} drop out on differentiation. If we make this formal definition of $\sigma'(x)$, then for $k \in I$, note that

$$\sigma'(\alpha^{-k}) = \alpha^k \prod_{\substack{i \in I \\ i \neq k}} (\alpha^{i-k} - 1).$$

So, from (3.10), we find

$$e_k = \begin{cases} 0, & \sigma(\alpha^{-k}) \neq 0; \\ \alpha^{-k(L-1)} \omega(\alpha^{-k}) / \sigma'(\alpha^{-k}), & \sigma(\alpha^{-k}) = 0. \end{cases} \tag{3.11}$$

Often $L = 1$ is chosen to simplify the expression.

The process of finding the roots of the error-locator polynomial is often performed by a *Chien search* [17], which involves evaluating $\sigma(x)$ iteratively at consecutive powers of α and testing for zero. Although this method constitutes an exhaustive search, it can be implemented fairly efficiently in hardware, as we shall see in Chapter six; for decoding applications in which the codeword components are received sequentially, almost no penalty is paid for such a brute force technique. In binary BCH codes, where an error is known to have value 1, the error locations alone are sufficient to specify the error values; in this case only a Chien search is needed. The more general expression for error values (3.11), known as the *Forney algorithm* [26], must be used for non-binary codes. Although both a Chien search and the Forney algorithm are required for Reed-Solomon codes, we shall often refer to the combined process as a Chien search, for simplicity.

3.6 Performance of Reed-Solomon Codes

In his doctoral thesis, Cohen [19] presents several criteria for determining when the use of Reed-Solomon codes is appropriate and how their features can be utilized most effectively. The first conclusion is that, for codes of similar block lengths and redundancy, RS codes have distinct advantages with respect to binary codes or convolutional codes only on channels in which errors tend to occur in bursts, even when interleaving is utilized to enhance the burst protection of the non-RS codes. [7]. However, if the channel is subjected to noise which is random from bit to bit, a binary BCH code offers better error protection than an RS code or a non-binary BCH code. Because many channels of interest are characterized by error bursts, Reed-Solomon codes are

therefore of great practical importance. Another disadvantage of binary codes is that, to achieve the same total block length (in bits) as a Reed-Solomon code, a binary BCH code is typically built over a larger field, implying the need for a wider arithmetic unit in the decoder.

Cohen's second conclusion is that high-rate Reed-Solomon codes are adequate for almost all applications. If channel error rates dictate the need for a low-rate code, a *concatenated* coding scheme, as shown in Figure 3-4, will usually outperform a straight RS code of the same rate. In a concatenated system, there are two codes, an inner code and an outer code: the outer code is typically a high-rate RS code, while the inner code can be a low-rate code which is much simpler to encode and decode. The overall coding rate of the system is the product of the inner and outer code rates. Because the area-time complexity for a Reed-Solomon decoder is at least on the order of the square of the redundancy [24], the cost of the concatenated system is usually lower than that of a corresponding low-rate RS code system.

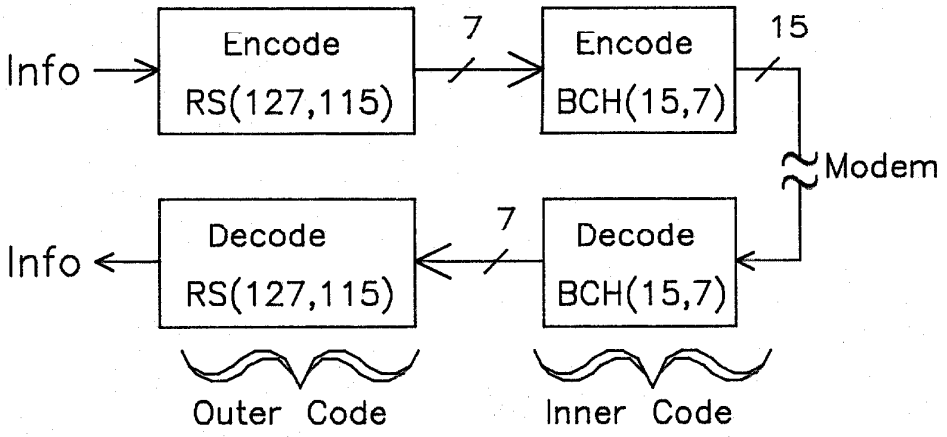

Figure 3-4. Example of a Concatenated Coding Scheme

As an example, consider the concatenated system shown in Figure 3-4. The inner code is a (15,7) binary BCH code, which is capable of correcting two random bit errors in each block of fifteen bits; the output of the inner decoder consists of seven bit blocks, which could serve as the input to a (127,115) RS code over GF(128), capable of correcting six character errors of up to seven bits each. While the overall rate is roughly 0.42, each of these codes is fairly easy to implement. In particular, the BCH code can be both encoded and decoded using lookup tables. Also, if soft information is available from the channel, perhaps the inner code could be a convolutional code; this type of concatenated system is very powerful, utilizing the best features of convolutional and block codes.

For a given channel, how does one go about selecting the appropriate combination of block length, rate, and interleaving depth to meet the reliability and/or cost objectives for the system? As an example of the decisions which must be confronted, it is possible to correct bursts either by adding additional redundant characters or by interleaving to a greater depth. Consider the following shortened RS codes over GF(256): a (250,230) non-interleaved code versus a (125,115) code interleaved to depth two. Both codes have the same number of redundant characters, the same rate, and the same *interleaved* block length (250 bytes), and each code can correct bursts

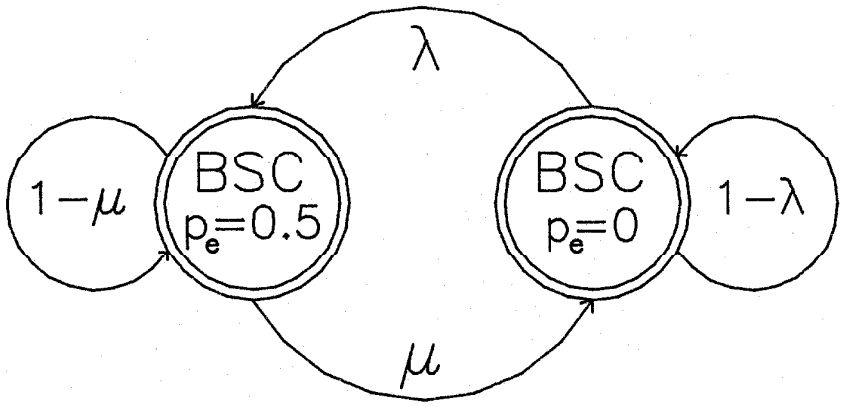

Figure 3-5. The Gilbert Channel Model

up to eighty bits long. The first code can correct more error patterns than the interleaved code but requires a more complex decoder; thus, interleaving has both advantages and disadvantages. In general, for a given block length and rate, a non-interleaved code is more powerful than an interleaved code, but block length limitations on RS codes and the hardware costs for additional redundancy must be taken into account.

The tradeoffs involved also obviously depend on the particular channel model selected. Perhaps the simplest channel model is known as the *binary symmetric channel* or BSC, in which every bit transmitted has a probability p_e of being received in error; this model clearly corresponds to random errors. Often, curves of output bit error probability versus input bit error probability are plotted for Reed-Solomon codes based on a BSC channel model, mainly because the result can be obtained in closed form. However, as mentioned above, RS codes perform best in a bursty noise environment, so using the BSC model is not particularly appropriate. Employing a slightly more complex channel model, known as the Gilbert model [27], exact numerical results can be obtained for the RS coding scheme of choice (see Appendix B for details). The Gilbert model, illustrated in Figure 3-5, is essentially a Markov process. One state is known as the good channel, in which all transmitted bits are received without error (i.e., $p_e = 0$). Transitions occur from the good channel to the bad channel in a discrete Poisson process with probability λ at each bit. The bad channel, a BSC with $p_e = 0.5$, corresponds to a burst in progress, and the burst continues with Poisson statistics, ending with probability μ at each bit. For $\mu = 1.0$ and λ small, the Gilbert model closely approximates a BSC with bit error probability $\lambda/2$. Obviously, the average spacing between bursts is $1/\lambda$ bits, while the average burst length is $1/\mu$ bits; these two parameters are often known for the channel of interest, so an analysis can be performed to predict the probability that the decoder will encounter a block which is not decodable. However, no real channel can be modelled exactly by such a simple model, so the results of such analyses must be interpreted somewhat qualitatively.

In Figure 3-6, the contours for constant output bit error probability using the Gilbert model are plotted for a (255,239) RS code over GF(256), interleaved to depth $d = 2$ and $d = 4$. Observe that the effect of interleaving is essentially to scale the burst length axis, as one might expect. In fact, when the average burst size is exactly one bit, all errors are random, so there is no difference between the two plots at this point. For small average burst size (i.e., smaller than

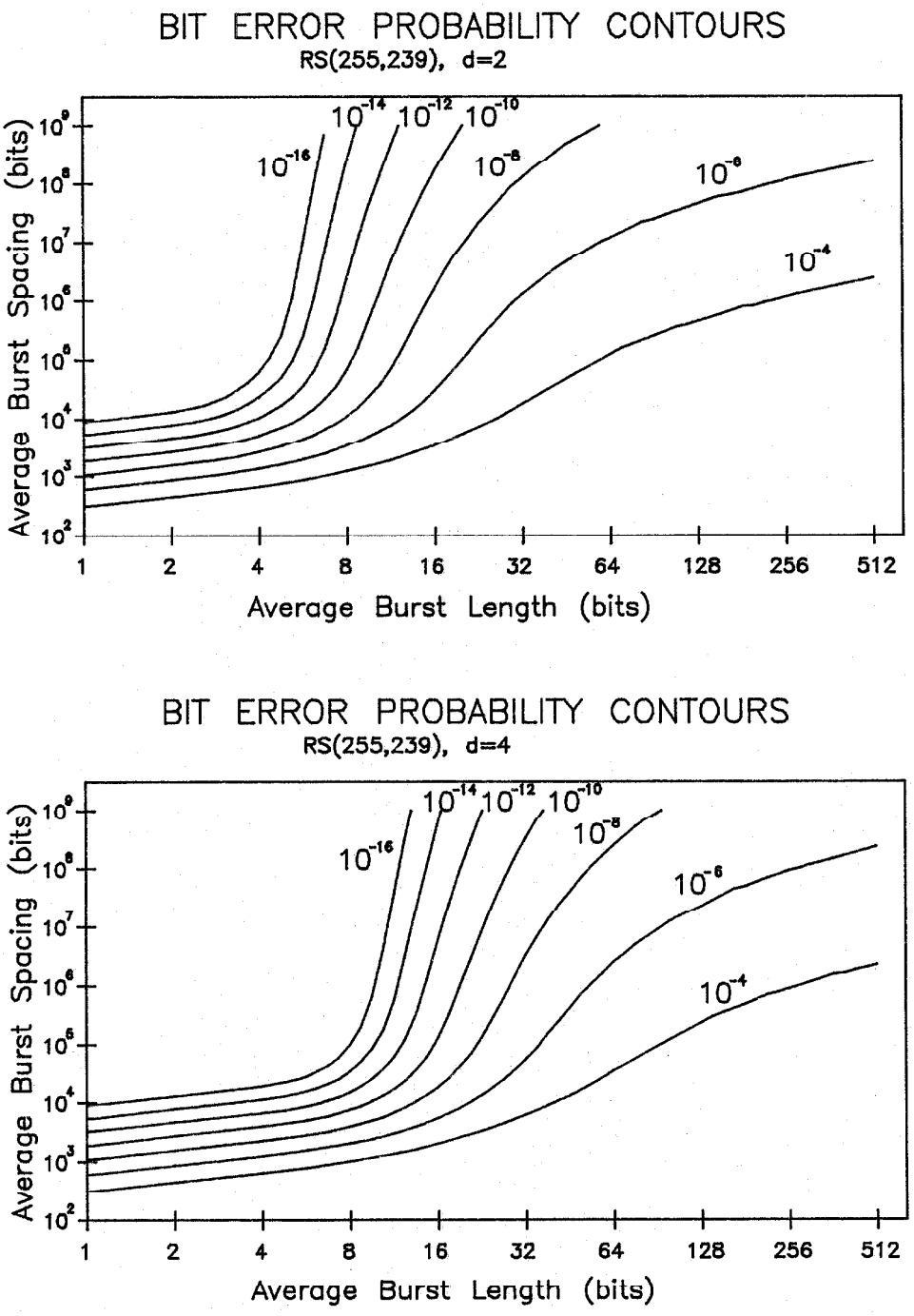

Figure 3-6. Gilbert Model Error Statistics For Interleaved RS Codes

the character size), the probability of decoder error increases only slightly with increasing burst length. Then at some critical point the error probability increases rapidly with burst length. As the average burst length becomes large enough for a single burst to cause an uncorrectable error pattern, the curve flattens out again; in other words, the bit error probability depends much more heavily on whether any burst occurs in the block. A fair amount of numerical computation is involved in generating these plots, but such curves can be very helpful in comparing the tradeoffs in coding systems. Appendix B contains details of the computations as well as an entire family of curves for various output bit error probabilities and depths of interleaving.

As with many engineering decisions, the selection of coding parameters is as much an art as a science, although many analytic and numerical methods can assist in the design process. Even given a flexible single-chip Reed-Solomon decoder, many subtle system factors can influence the choice of the appropriate code, but there is little doubt that the tradeoffs involved can be greatly simplified by a VLSI decoder implementation.

Chapter 4

Bit-Serial Multiplication

4.1 Motivation

Bit-serial multiplication over finite fields has recently been given much attention in the literature. Berlekamp [5] has shown how to build efficient Reed-Solomon encoders using a dual-basis bit-serial multiplier, and implementations of this technique have been realized in both conventional logic [5] and LSI chips [32]. Upon contemplating this result, it becomes apparent that the major conceptual advance is in the bit-serial multiplication method; the encoder is but one particular application. For decoder systems, our hope is to fit many multipliers on a single integrated circuit, molding the chip architecture directly to the decoding algorithm of interest.

Although Berlekamp applied his technique only to multiplication by known constants, it can readily be generalized to the product of two arbitrary field elements, as we will see in the following sections. Omura and Massey [39] have presented another scheme of performing bit-serial multiplication, using a basis consisting of all conjugates of a given field element. The question then naturally arises: which of these approaches is more efficient? To our knowledge, no one has yet addressed the issue of choosing the representation of the field elements which (in some sense) minimizes the cost of building a bit-serial multiplier.

In this chapter we shall investigate the structure of finite fields, seeking to optimize bit-serial computation. A family of bit-serial multipliers which includes Berlekamp's and Omura's approaches as special cases will be derived; among all these methods, the dual-basis structure is shown to be superior in several respects. Further, it is proved that a *self-dual* basis can be found if and only if some elements of the field have a trinomial as their minimal polynomial, which result unfortunately excludes GF(256). Also, a more traditional approach to multiplication will be presented, which employs a canonical basis and is analagous to the integer shift-and-add method. We also demonstrate how multiplicative inversion can be accomplished in a bit-serial fashion, greatly reducing the complexity traditionally associated with this operation. Each of these results can impact a decoder architecture significantly.

4.2 Area-Time Tradeoffs in Multiplier Design

The following sections will examine in great detail how to build multipliers of a particular type over GF(2^m). Before plunging into the algebra, it will be instructive to give some rationale for

selecting bit-serial arithmetic in general and this approach in particular. Much study has been done on bit-serial arithmetic methods for the integers, and computations over GF(2^m) involve many similar features. However, a distinguishing feature of finite-field multiplication is that both factors must be completely available before any part of the result can be produced. In other words, there is no concept of least significant or most significant bits in the representation of a field element, so integer multiplication techniques must be reinterpreted in light of this fact if any correspondence is to be made.

In the past, the complexity of a decoding algorithm has often been measured by the number of finite-field multiplications required, because the design of high-performance decoding systems has typically revolved around a single, extremely fast, fully parallel multiplier [3,19]. While it is quite feasible to build a parallel multiplier on a VLSI chip, the structures involved are rather unwieldy. Multiplication of two elements of GF(2^m) requires m^2 Boolean AND operations, together with at least $m^2 - 1$ XOR operations. The throughput of such a multiplier would vary considerably, depending on the amount of pipelining and the structure of the parity computation (see Table 4-1 below), but it is clear that wiring will contribute considerably to the area, limiting the number of such multipliers that could fit on a single chip.

In VLSI, the complexity of communication is often a more significant determinant of chip area than the overall count of computational operations. Using a small number of parallel multipliers in a decoder chip is inefficient for two reasons. First, each multiplier will typically be multiplexed between many inputs and/or outputs, and the cost of such sharing, including the communication of operands and results, may well outweigh the cost of the multiplier itself. More importantly, the polynomial manipulations involved in decoding algorithms have a high degree of inherent parallelism. By executing many multiplications concurrently, even if each operation is (individually) relatively slow, appreciable gains in throughput can be achieved.

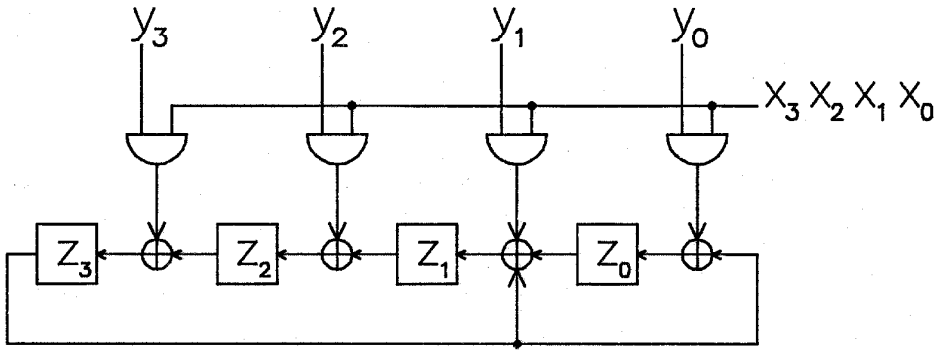

Figure 4-1. A Shift-and-Add Multiplier over GF(16)

Bit-serial arithmetic presents the possibility of fitting large numbers of multipliers on a decoder chip. There are two well-known approaches to bit-serial multiplication over finite fields. The first technique is somewhat analagous to the integer shift-and-add method [4, Section 2.42], in that one factor must be completely available throughout the procedure, while the other factor is made available one bit at a time. Such asymmetry introduces a difference in the latency of the

Multiplier Type	Parity Structure	A	T	p	T/p	AT/p
Parallel	pipelined	m^2	$\log m$	$\log m$	1	m^2
Parallel	tree	m^2	$\log m$	1	$\log m$	$m^2 \log m$
Parallel	chain	m^2	m	1	m	m^3
Shift-and-Add	(none)	m	m	1	m	m^2
Bit-serial	pipelined	m	m	1	m	m^2
Bit-serial	tree	m	$m \log m$	1	$m \log m$	$m^2 \log m$
Bit-serial	chain	m	m^2	1	m^2	m^3

Table 4-1. Order Estimates for GF(2^m) Multiplier Structures

result with respect to the two factors, which may have an impact on higher level system timing. The shift-and-add method (see Figure 4-1) is attractive because no parity tree is required, and both factors can be expressed in a canonical basis. Also, multiplication by fixed constants can be *hard-wired* into the circuitry. This technique in some sense is a hybrid between parallel and bit-serial computation: the result is obtained in parallel and may then be shifted out serially.

In the second approach, both factors are required throughout the computation, producing one bit of the result each clock cycle [5, 39]. This method is perhaps more strictly bit-serial, in that the result is actually produced sequentially instead of being serialized after the fact; thus it may be attractive for systems in which all communication is to be done bit-serially, since the latency is a constant m with respect to both factors. As we shall see below, however, performing multiplication in this fashion in general also requires area proportional to m^2, unless great care is taken in choosing a representation for the field elements.

In Table 4-1, we present order estimates of the area A, latency T, and number of multiplication problems p that can be handled concurrently for some of the approaches discussed above. It is very important to realize, however, that for coding applications we are interested in small m (say $m < 13$); therefore, asymptotic behavior may not be at all indicative of the proper performance tradeoffs. For example, a parity tree can replace a time factor of $O(m)$ by $O(\log m)$, but the wiring overhead could be quite costly for small m, so the constants in front of the order estimates are crucial. In MOS technology, a linear parity chain can be implemented entirely with pass gates, while a tree structure requires restoring stages, which can be costly in area, time, and power. Table 4-1 does not attempt to give the constants which accompany the actual order, since these values will depend on the particular implementation technology, but for small m the difference between $O(m)$ and $O(\log m)$ depends crucially upon these values.

It is clear that there exist intermediate approaches to performing multiplication, such as producing two bits of the result at a time, or four, etc., and it is quite possible that one of these methods will optimize some area-time metric. However, bit-serial communication on a decoder chip will already be expensive enough in terms of wiring, and efficient utilization of multi-bit arithmetic units requires multi-bit communication channels. Further, our aim is to fit as many multipliers on a chip as possible, and bit-serial methods will obviously minimize the area required. The rest of this chapter will be devoted to a study of bit-serial multipliers of the two types discussed above. First we will treat the case in which the bits of the product are produced sequentially; then the inherently simpler shift-and-add multiplier will be generalized by analogy to the former case.

4.3 Resolving Field Elements into Basis Components

Before discussing bit-serial multipliers, we need some tools for extracting the bit representation of field elements in a given basis. Suppose we have a basis $B = \{\mu_0, \mu_1, \ldots, \mu_{m-1}\}$ for GF(2^m) over GF(2). In other words, every element z of the field can be uniquely expressed as a linear combination of the basis elements

$$z = \sum_{i=0}^{m-1} z_i \mu_i$$

where each $z_i \in GF(2)$.

To find the components of an arbitrary field element z in this basis, we seek a set of functions $\{f_j\}$ such that $f_j(z) = z_j$. Note that each f_j is a linear function from GF(2^m) into GF(2), since

$$f_j(x+y) = x_j + y_j = f_j(x) + f_j(y).$$

Thus, using the results of Section 2.6, we know that there exists a set $D = \{\alpha_0, \alpha_1, \ldots, \alpha_{m-1}\}$ such that

$$f_j(z) = z_j = \mathrm{Tr}(z\alpha_j).$$

Now, since trace is a linear function, we know that

$$\mathrm{Tr}(z\alpha_j) = \mathrm{Tr}\left(\sum_{i=0}^{m-1} z_i \mu_i \alpha_j\right) = \sum_{i=0}^{m-1} z_i \mathrm{Tr}(\mu_i \alpha_j),$$

which can hold if and only if

$$\mathrm{Tr}(\mu_i \alpha_j) = \delta_{i,j}\,. \tag{4.1}$$

Clearly, D also forms a basis for GF(2^m), since if

$$0 = \sum_{j=0}^{m-1} c_j \alpha_j$$

then for each i

$$0 = \mathrm{Tr}\left(\mu_i \left(\sum_{j=0}^{m-1} c_j \alpha_j\right)\right) = \sum_{j=0}^{m-1} c_j \mathrm{Tr}(\mu_i \alpha_j) = c_i.$$

D is known in the literature as the *dual basis* of B [5,36].

The dual basis D thus provides one method for determining the components of z in the basis B. From (4.1), by analogy with the unit vectors in three-space, we see that the dual basis is in some sense orthogonal to the basis B. In other words, to pick out a specific component z_j, apply an inner product of the form $\mathrm{Tr}(z\alpha_j)$. However, for circuit implementation, it will be more convenient to have an iterative scheme of generating the z_j. Given the basis B, we can extract

the components of z one at a time using a non-singular linear transformation T such that

$$\begin{aligned} z_0 &= \mathrm{Tr}(\alpha_0 z) \\ z_1 &= \mathrm{Tr}(\alpha_1 z) = \mathrm{Tr}(\alpha_0 T(z)) \\ z_2 &= \mathrm{Tr}(\alpha_2 z) = \mathrm{Tr}(\alpha_0 T^2(z)) \\ &\vdots \\ z_{m-1} &= \mathrm{Tr}(\alpha_{m-1} z) = \mathrm{Tr}\big(\alpha_0 T^{m-1}(z)\big), \end{aligned} \tag{4.2}$$

where $T^j(z) = T\big(T^{j-1}(z)\big)$, and $T^0(z) = z$. Actually there are 2^{m-1} such transformations, specified by their results on the basis elements:

$$\begin{aligned} T(\mu_0) &= \mu_{m-1} \\ T(\mu_j) &= \mu_{j-1} + c_j\,\mu_{m-1}, \qquad \text{for} \quad j = 1, \ldots, m-1\,, \end{aligned} \tag{4.3}$$

where each $c_j \in \mathrm{GF}(2)$, with $c_0 = 1$. The c_j's may be chosen arbitrarily, and we will attempt to utilize this freedom below. Note that applying T to an element expressed in the basis B amounts to a shift operation, with a feedback term specified by the c_j's.

4.4 Choosing an Optimal Basis

Now suppose that we wish to find the product of two field elements, $z = xy$, where

$$x = \sum_{i=0}^{m-1} x_i \mu_i \qquad \text{and} \qquad y = \sum_{j=0}^{m-1} y_j \mu_j.$$

We implement the logic to produce the first bit of the result, namely

$$z_0 = \mathrm{Tr}(\alpha_0 z) = \mathrm{Tr}(\alpha_0 xy) = \sum_{i,j=0}^{m-1} x_i y_j \mathrm{Tr}(\alpha_0 \mu_i \mu_j). \tag{4.4}$$

Although this logic will be fairly involved, note that by applying T to the product successively, the remaining bits of the result are produced, reusing the same first bit logic.

In choosing the basis B, we have several goals. First, the transformation T must factor somehow, so that T may be applied to the product $z = xy$ by individually transforming x and y. Further, these individual transformations must be relatively simple to apply to x and y. Lastly, implementation of the function $\mathrm{Tr}(\alpha_0 xy)$ must be as simple as possible. Let us examine these (somewhat subjective) requirements in turn, with the caveat that it is unclear initially whether all our goals can be satisfied simultaneously.

4.5 Factorable Linear Transformations

The linear transformation T must not only satisfy equations (4.2), but it is also required to form some sort of homomorphism with respect to field multiplication. In particular, there must exist

functions R and S such that

$$\begin{aligned} \mathrm{Tr}(\alpha_0 T(xy)) &= \mathrm{Tr}(\alpha_0 R(x) S(y)) \\ \mathrm{Tr}(\alpha_0 T^2(xy)) &= \mathrm{Tr}(\alpha_0 R^2(x) S^2(y)) \\ &\vdots \\ \mathrm{Tr}(\alpha_0 T^{m-1}(xy)) &= \mathrm{Tr}(\alpha_0 R^{m-1}(x) S^{m-1}(y)), \end{aligned} \tag{4.5}$$

so that the bits of the product $z = xy$ will be correct. What form can these functions take? Since the problem is symmetric, let us concentrate on R. Each result below applies to S as well.

Lemma 4.1. $R(x_1) = R(x_2) \Leftrightarrow x_1 = x_2$. *In other words, R is one-to-one.*

Proof: Given two elements $u, v \in \mathrm{GF}(2^m)$ such that $R(u) = R(v)$, suppose $u \neq v$. Consider the two products $z = uy$ and $w = vy$, where y is an arbitrary nonzero element of $\mathrm{GF}(2^m)$. Looking at the bit representation of these products,

$$\begin{aligned} z = uy \mapsto (z_0, z_1, z_2, \ldots) &= \left(\mathrm{Tr}(\alpha_0 uy), \mathrm{Tr}(\alpha_0 R(u) S(y)), \mathrm{Tr}(\alpha_0 R^2(u) S^2(y)), \ldots\right) \\ w = vy \mapsto (w_0, w_1, w_2, \ldots) &= \left(\mathrm{Tr}(\alpha_0 vy), \mathrm{Tr}(\alpha_0 R(v) S(y)), \mathrm{Tr}(\alpha_0 R^2(v) S^2(y)), \ldots\right). \end{aligned}$$

Since $R(u) = R(v)$, then $R^j(u) = R^j(v)$, for all $j > 0$; thus $z_j = w_j$, for $j > 0$. But $z - w = uy - vy = (u - v)y \neq 0$, so we must have $z_0 \neq w_0$, implying

$$z - w \mapsto (1, 0, 0, \ldots) \mapsto \mu_0.$$

Or,

$$y = (z - w)(u - v)^{-1} = \mu_0 (u - v)^{-1}.$$

Now there is a contradiction, since we allowed y to be an arbitrary element of the field, yet we have shown it to be a fixed element. Therefore, we must have $u = v$. ∎

Corollary 4.2. *For each $z \in \mathrm{GF}(2^m)$, there exists an $x \in \mathrm{GF}(2^m)$ such that $R(x) = z$. In other words, R is onto.*

Proof: Since R is one-to-one mapping from a finite set into itself, it must also be onto. ∎

Lemma 4.3. $R(0) = 0$.

Proof: By Corollary 4.2, $\exists\, x \in \mathrm{GF}(2^m)$ such that $R(x) = 0$. Suppose $x \neq 0$. Let z be an arbitrary field element. Note that $z = xy$, where $y = zx^{-1}$. So,

$$z_1 = \mathrm{Tr}(\alpha_0 T(z)) = \mathrm{Tr}(\alpha_0 R(x) S(y)) = 0.$$

But we know that there exist field elements (e.g., $z = \mu_1$) for which $z_1 \neq 0$. So, by contradiction, we must have $x = 0$. ∎

Lemma 4.4. *R is a non-singular linear transformation.*

Proof: Note that for all $u, v, y \in \mathrm{GF}(2^m)$,

$$\begin{aligned}\mathrm{Tr}(\alpha_0 R(u+v)S(y)) &= \mathrm{Tr}(\alpha_0 T((u+v)y)) = \mathrm{Tr}(\alpha_0 T(uy)) + \mathrm{Tr}(\alpha_0 T(vy)) \\ &= \mathrm{Tr}(\alpha_0 R(u)S(y)) + R(v)S(y) = \mathrm{Tr}(\alpha_0 (R(u)+R(v))S(y)).\end{aligned}$$

By Corollary 4.2, since y is an arbitrary element of $\mathrm{GF}(2^m)$, both $S(y)$ and $\beta = \alpha_0 S(y)$ may also assume any field value. So, for all u, v, with β an arbitrary field element,

$$\mathrm{Tr}(\beta[R(u+v) - (R(u)+R(v))]) = 0.$$

This can happen only if the expression in brackets is identically zero. Then, for all $u, v \in \mathrm{GF}(2^m)$,

$$R(u+v) = R(u) + R(v)\,.$$

Thus, R is linear, and by the previous lemmas we know that it is non-singular. ∎

Lemma 4.5. *If R and S satisfy equations (4.5), then for all $x, y \in \mathrm{GF}(2^m)$,*

$$T(xy) = R(x)S(y) + \mu_{m-1}\mathrm{Tr}(xQ(y)), \tag{4.6}$$

where $Q(y)$ is a linear transformation.

Proof: Consider the representation of $z = T(xy)$ and the product $w = R(x)S(y)$ in the basis B. Using equations (4.5),

$$\begin{aligned}w_0 &= \mathrm{Tr}(\alpha_0 w) = \mathrm{Tr}(\alpha_0 R(x)S(y)) = \mathrm{Tr}(\alpha_0 T(xy)) = z_0 \\ w_1 &= \mathrm{Tr}(\alpha_0 T(w)) = \mathrm{Tr}(\alpha_0 R^2(x)S^2(y)) = \mathrm{Tr}(\alpha_0 T^2(xy)) = z_1 \\ &\vdots \\ w_{m-2} &= \mathrm{Tr}(\alpha_0 T^{m-2}(w)) = \mathrm{Tr}(\alpha_0 R^{m-1}(x)S^{m-1}(y)) = \mathrm{Tr}(\alpha_0 T^{m-1}(xy)) = z_{m-2}.\end{aligned}$$

In other words, z and w are identical in their first $m-1$ components, so either $z = w$ or $z = w + \mu_{m-1}$. Then there exists a function $g(x, y)$ which takes on values in GF(2) such that

$$T(xy) = R(x)S(y) + \mu_{m-1} g(x, y).$$

If we fix y, then $g(x, y) = g_y(x)$, and, because both T and R are linear,

$$g_y(u+v) = g_y(u) + g_y(v),$$

for all $u, v \in \mathrm{GF}(2^m)$. Thus g_y is a linear function from $\mathrm{GF}(2^m)$ into GF(2), so it must have the form $g_y(x) = \mathrm{Tr}(xQ(y))$. By similar argument, for all $u, v, x \in \mathrm{GF}(2^m)$,

$$\mathrm{Tr}(xQ(u+v)) = \mathrm{Tr}(xQ(u)) + \mathrm{Tr}(xQ(v)).$$

Therefore, $Q(y)$ is a linear function. ∎

At first glance it would appear that equation (4.6) is not symmetric between x and y. However, from Section 2.6 we know that any linear function over GF(2^m) has the form

$$Q(y) = \sum_{j=0}^{m-1} q_j y^{2^j}.$$

Then, since $\mathrm{Tr}(x) = \mathrm{Tr}\left(x^{2^j}\right)$, we deduce that

$$\mathrm{Tr}(xQ(y)) = \mathrm{Tr}\left(x \sum_{j=0}^{m-1} q_j y^{2^j}\right) = \mathrm{Tr}\left(y \sum_{i=0}^{m-1} (xq_i)^{2^{m-i}}\right).$$

So indeed the form is still symmetric between x and y.

Lemma 4.6. *$Q(y) = \gamma y$, for some $\gamma \in$ GF(2^m).*

Proof: Expressing the linear transformations of equation (4.6) in their polynomial forms (see Section 2.6), we find

$$\sum_{i=0}^{m-1} t_i x^{2^i} y^{2^i} = \sum_{i,j=0}^{m-1} r_i s_j x^{2^i} y^{2^j} + \mu_{m-1} \sum_{i,j=0}^{m-1} q_j^{2^i} x^{2^i} y^{2^{i+j}}.$$

This equation must hold identically for all x and y, so we have a matrix of coefficient relations to be satisfied. In particular, we may break these conditions into diagonal terms (i.e., $i = j$)

$$t_i = r_i s_i + \mu_{m-1} q_0^{2^i}, \quad \text{for} \quad i = 0, \ldots, m-1 \tag{4.7}$$

and off-diagonal terms

$$r_i s_j = \mu_{m-1} q_{j-i}^{2^j}, \quad \text{for} \quad i \neq j, \tag{4.8}$$

where all indices are understood to be taken modulo m.

We want to show that all the off-diagonal terms of q are zero. Note from equation (4.8) that if even one such term is zero, they are all zero. For example, if $r_i = 0$, then for all $j \neq i$, $q_{j-i} = 0$. So let us assume that $r_i \neq 0$ and $s_i \neq 0$ for all i, and that $q_i \neq 0$ when $i \neq 0$; we will look for a contradiction.

First, we note that $R(x)$ is defined only up to a scale factor, which may be absorbed into $S(y)$, so since $r_0 \neq 0$ we may assume without loss of generality that $r_0 = \mu_{m-1}$. Then, using equation (4.8) with $i = 0$ we find

$$s_j = q_j, \quad \text{for} \quad j \neq 0. \tag{4.9}$$

Let us define $v_i = \mu_{m-1}^{-1} r_i$, so that $v_0 = 1$. Now consider the case $i = 1$:

$$v_1 q_j = q_{j-1}^2, \quad \text{for} \quad j = 2, 3, \ldots, m-1.$$

So, $v_1 = q_1^2/q_2$. Since every element of GF(2^m) has a unique square root, we may choose $\beta \in$ GF(2^m) such that $\beta^2 = q_2/q_1$. Proceeding along the row, we find

$$\begin{aligned} q_3 &= (q_2/q_1^2)q_2^2 = q_2\beta^4 \\ q_4 &= (q_2/q_1^2)q_3^2 = q_2\beta^{12} \\ &\vdots \end{aligned}$$

and in general, for $j > 2$,

$$q_j = (q_2/q_1^2)q_{j-1}^2 = q_2\beta^{2^j-4}. \tag{4.10}$$

In fact, equation (4.10) also holds for $j = 1, 2$, so from (4.9)

$$s_j = q_2\beta^{2^j-4}, \quad \text{for} \quad j = 1, 2, \ldots, m-1.$$

What about s_0 ? We know from (4.8) that

$$s_0 = q_{m-1}^2/v_1 = \left(q_2\beta^{2^{m-1}-4}\right)^2\beta^4/q_2 = q_2\beta^{2^m-4} = q_2\beta^{2^0-4},$$

since $\beta^{2^m} = \beta$, for all $\beta \in$ GF(2^m). Thus,

$$s_j = q_2\beta^{2^j-4}, \quad \text{for} \quad j = 0, 1, 2, \ldots, m-1,$$

and

$$S(y) = \sum_{j=0}^{m-1} s_j y^{2^j} = q_2\beta^{-4}\sum_{j=0}^{m-1}(\beta y)^{2^j} = q_2\beta^{-4}\mathrm{Tr}(\beta y).$$

By Lemma 4.4, $S(y)$ is non-singular, yet $\mathrm{Tr}(\beta y)$ is singular for any value of β. By contradiction, all of the off-diagonal terms must be zero, so $Q(y) = q_0 y$. ∎

Theorem 4.7. *If R, S, and T satisfy equations (4.5), then*

$$T(z) = \alpha z^{2^k} + \mu_{m-1}\mathrm{Tr}(\gamma z), \tag{4.11}$$

for some $\alpha, \gamma \in$ GF(2^m), $\alpha \neq 0$, with $0 \leq k \leq m-1$, and $\mathrm{Tr}(\gamma\mu_0) = 0$.

Proof: By the previous lemma, $r_i s_j = 0$ whenever $i \neq j$. Since R and S are both non-singular, however, at least one term in each must be nonzero. Both requirements can be satisfied only if there exists a k such that $r_k \neq 0$, $s_k \neq 0$, and $r_i = s_i = 0$ for $i \neq k$. In other words,

$$R(x)S(y) = r_k s_k x^{2^k} y^{2^k} = \alpha z^{2^k},$$

where $\alpha = r_k s_k \neq 0$ and $z = xy$.

Also, from equation (4.3) we know that $T(\mu_0) = \mu_{m-1}$, or

$$\mu_{m-1} = \alpha\mu_0^{2^k} + \mu_{m-1}\mathrm{Tr}(\gamma\mu_0).$$

Note that if $\mathrm{Tr}(\gamma\mu_0) = 1$, then $\alpha\mu_0^{2^k} = 0$, which is impossible since both terms are nonzero. Thus, $\mathrm{Tr}(\gamma\mu_0) = 0$, and $\mu_{m-1} = \alpha\mu_0^{2^k} = T(\mu_0)$. ∎

In conclusion, we have found that $T(z)$ must have the form of a constant multiplying some conjugate of z, plus a feedback term. There are exactly 2^{m-1} constants γ which satisfy the requirements of Theorem 4.7, and it is evident that γ may be chosen to give any desired feedback terms c_j in equation (4.3). The linear transformations R and S have the form of a constant multiplying the same conjugate of z, without the additional feedback terms. It should be emphasized at this point that our result is the most general form of a linear transformation satisfying equations (4.5), since no additional assumptions have been made in the lemmas leading up to Theorem 4.7.

4.6 Applying the Transformations

Having derived the general form of the transformations involved in bit-serial multiplication, let us examine how efficiently they can be implemented in hardware. Notice that if we choose $\gamma = 0$ in equation (4.11), all three transformations, R, S, and T, have the same form. Thus, it is evident that there exists a basis in which applying R amounts to a shift operation similar to (4.3), and this basis is in some sense the natural basis to choose for applying R. A similar basis can be found for S, and by expressing x and y in their respective natural bases, the implementation will be (arguably) optimized.

Several questions arise at this point. First, since equation (4.11) is only a necessary condition on the form of T, what values of α_0, α, γ, and k will actually produce a linearly independent set B? Also, given a transformation T, what is the procedure used to find the basis B, and what is the form of the resulting basis? Before proceeding, it will be instructive to work out in detail a few specific examples using Theorem 4.7. We shall find that the two generally known methods of bit-serial multiplication occur as special cases of equation (4.11).

Example 4.1: The Canonical Dual Basis

Suppose we choose $k = 0$ and $\gamma = 0$ in equation (4.11). Then $T(z) = \alpha z$. Given an α_0, from equations (4.2) we require

$$\mathrm{Tr}(\alpha_j z) = \mathrm{Tr}\big(\alpha_0 T^j(z)\big) = \mathrm{Tr}\big(\alpha_0 \alpha^j z\big).$$

In other words, $\alpha_j = \alpha_0 \alpha^j$. For the set $\mathrm{D} = \{\, \alpha_j \mid j = 0, 1, \ldots, m-1 \,\}$ to form a basis, it is clear that α must satisfy an irreducible polynomial of degree m over GF(2). Then the basis D is just a constant multiple of the canonical basis formed by the first m powers of α, and given D it is a straightforward (though tedious) task to compute its dual basis B, which is the natural basis for T. When an element z is expressed in the basis B, applying T (i.e., multiplying by α) amounts to a shift of the register containing z, with feedback terms specified by the nonzero coefficients in the minimal polynomial of α.

Berlekamp [5] has advocated using the canonical dual basis in Reed-Solomon encoders, where all the multiplications involve fixed constants. This type of basis is particularly attractive in such cases because we can define $R(x) = T(x) = \alpha x$ and $S(y) = y$, where y is the fixed constant. Since only the identity transformation is ever applied to y, there is no need to hold the constant in a register; instead the value of y can be hard-wired into the logic to produce $\mathrm{Tr}(\alpha_0 xy)$. Clearly x should be expressed in the basis B so that T can be applied very simply. If we express y in the canonical basis (instead of the dual basis),

$$y = \sum_{i=0}^{m-1} y_i \alpha^i = \sum_{i=0}^{m-1} y_i \alpha_0^{-1} \alpha_i ,$$

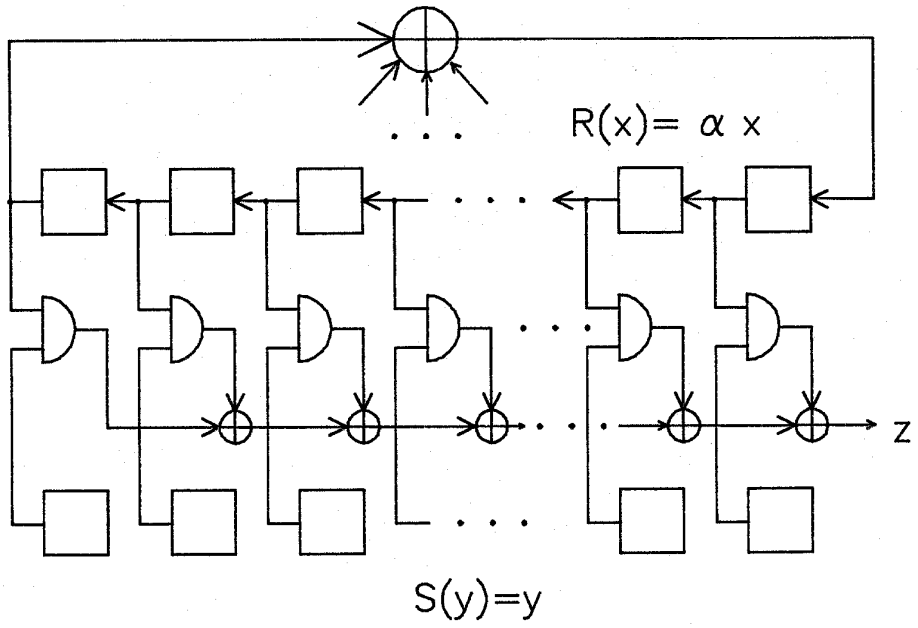

Figure 4-2. Dual-Basis Multiplier

then the expression for the first bit of the product simplifies:

$$\mathrm{Tr}(\alpha_0 xy) = \sum_{i,j=0}^{m-1} x_j y_i \mathrm{Tr}(\alpha_0 \alpha^i \mu_j) = \sum_{i,j=0}^{m-1} x_j y_i \mathrm{Tr}(\alpha_i \mu_j) = \sum_{i=0}^{m-1} x_i y_i.$$

In other words, the hardware takes the parity of the bits of x for which y_i is nonzero. When y is a constant, this subset of the bits of x can be hard-wired into the parity circuit. However, for y an arbitrary field element, the corresponding bits of x and y must be ANDed together, and the parity over all m bits produced; note the regular structure and locality in Figure 4-2. ∎

With one rather simple example under our belt, the procedure for producing a basis from the transformation given by (4.11) is now clear. Given an α_0 and a transformation T, we choose α_j such that

$$\mathrm{Tr}(\alpha_j z) = \mathrm{Tr}(\alpha_0 T^j(z)). \tag{4.12}$$

If the set $\mathrm{D} = \{\, \alpha_j \mid j = 0, 1, \ldots, m-1 \,\}$ forms a basis, then its dual basis B is the natural basis for T; otherwise T has no such basis. It should be noted that the value of γ in (4.11) has no bearing on the α_j, because according to (4.3), for $j < m$, any terms of $T^j(z)$ involving γ are in the subspace spanned by $\{\, \mu_j \mid j = 1, 2, \ldots, m-1 \,\}$, and we know by (4.1) that $\mathrm{Tr}(\alpha_0 \mu_j) = 0$ for $j \neq 0$. So in fact we may generally assume that $\gamma = 0$. With this simplification let us attempt a slightly more involved case.

Example 4.2: The Normal Basis

For $k \not\equiv 0$, we note that

$$\begin{aligned}
\mathrm{Tr}(\alpha_1 z) &= \mathrm{Tr}\left(\alpha_0 \alpha z^{2^k}\right) = \mathrm{Tr}\left(z(\alpha_0 \alpha)^{2^{k(m-1)}}\right) \\
\mathrm{Tr}(\alpha_2 z) &= \mathrm{Tr}\left(\alpha_0 \alpha^{1+2^k} z^{2^{2k}}\right) = \mathrm{Tr}\left(z(\alpha_0 \alpha^{1+2^k})^{2^{k(m-2)}}\right) \\
&\vdots
\end{aligned}$$

and in general we find

$$\alpha_j = \left(\alpha_0 \alpha^{(2^{jk}-1)/(2^k-1)}\right)^{2^{k(m-j)}}. \tag{4.13}$$

Now suppose that there exists an element $\omega \in \mathrm{GF}(2^m)$ such that $\omega^{2^k-1} = \alpha$. This will hold for all α when $\gcd(m,k) = 1$, because the only element of $\mathrm{GF}(2^m)$ which satisfies the polynomial $x^{2^k-1} - 1 = 0$ is $x = 1$; but for $\gcd(m,k) > 1$ only certain values of α satisfy this requirement. Given such a α,

$$\alpha_j = \left(\alpha_0 \omega^{2^{jk}-1}\right)^{2^{k(m-j)}} = \omega(\alpha_0/\omega)^{2^{k(m-j)}}. \tag{4.14}$$

When k and m are relatively prime, we see that the set $\mathrm{D} = \{\alpha_j \mid j = 0, 1, \ldots, m-1\}$ is just a constant multiple of the conjugates of α_0/ω, but when $\gcd(m,k) \not\equiv 1$, the set D contains duplicate elements and thus cannot be a basis.

If all conjugates of a given element are linearly independent, the resulting basis is said to be *normal*. So if D is a basis, then its dual basis is also a constant multiple of a normal basis, since if

$$\mathrm{Tr}(\mu_0 \alpha_j) = \mathrm{Tr}\left(\mu_0 \omega (\alpha_0/\omega)^{2^{k(m-j)}}\right) = \delta_{0,j}$$

then

$$\mathrm{Tr}\left((\mu_0\omega)^{2^{k(m-i)}} (\alpha_0/\omega)^{2^{k(2m-j-i)}}\right) = \delta_{i,j+i},$$

and the dual basis is

$$\mu_i = \omega^{-1}(\mu_0\omega)^{2^{k(m-i)}}, \tag{4.15}$$

which has the same form as D. Omura [39] has suggested the use of such a basis for bit-serial multiplication, with $\alpha = \omega = 1$ and $k = 1$, so that

$$T(z) = R(z) = S(z) = z^2.$$

In this case all the natural bases are identical, and the transformation is implemented as a circular rotation of the bits. However, in contrast to the previous example, fixed constants must be transformed and thus cannot be hard-wired into the circuitry. Further, note the lack of regularity and locality in the wiring of Figure 4-3, which uses a normal basis over GF(16) consisting of all primitive fifth roots of unity. ∎

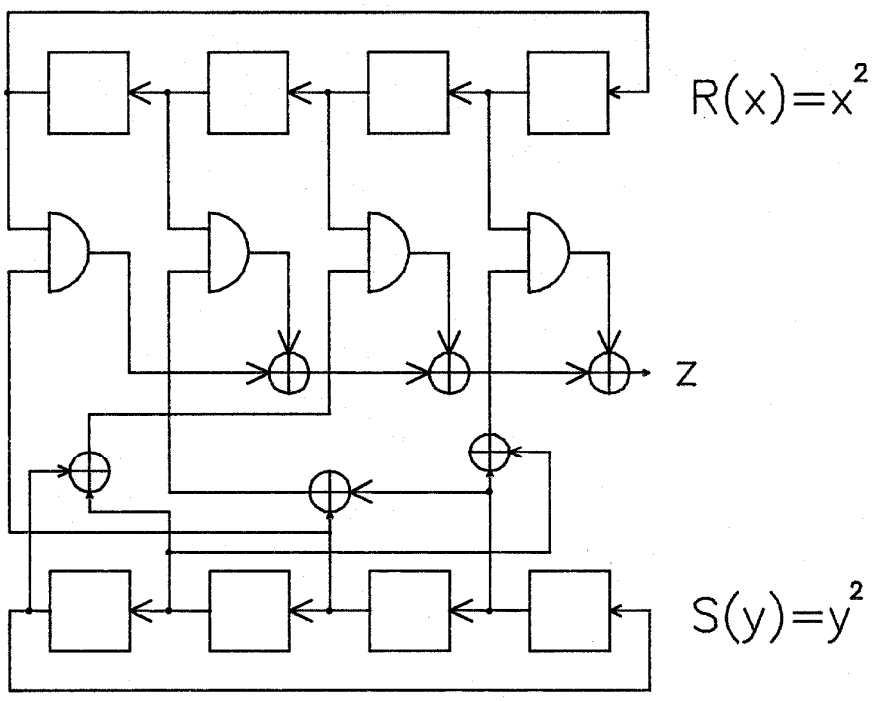

Figure 4-3. Normal-Basis Multiplier over GF(16)

There are many other transformations which produce bases that are not as easily characterized, and, using the above technique, a fairly efficient algorithm (roughly of complexity $2^{1.5m}$) can be devised to enumerate classes of bases which are equivalent up to scalar multiplication. However, multiplications by fixed constants are needed in all decoding algorithms, and the considerations below will further show that the canonical dual basis is the best choice for the problem at hand, so additional examples at this point are not needed.

In selecting a basis, it is also important to understand how a bit-serial multiplier will be used on a chip. In each of the known decoding algorithms, at least one polynomial operation consists of performing a multiplication involving the local coefficient and accumulating the product with another value to form the new coefficient. In other words, if the local values are expressed in a basis different from B (the basis in which the product is expressed), addition can no longer be done in a simple bit-serial fashion. Instead, the product will have to be converted to the local basis before the accumulation can take place. While it is conceivable that the basis change could be done efficiently in a bit-serial fashion, the extra circuitry involved requires at least an additional register, and this area penalty would be multiplied by the number of polynomial coefficients.

We therefore make the stipulation that the natural basis for R (or S) be the same as for T, which can happen only if $\gamma = 0$ in equation (4.11). Given this condition,

$$T(z) = R(z)S(1) \qquad \text{and} \qquad T(z) = R(1)S(z). \tag{4.16}$$

From (4.16) it can be shown that $T(xy) = T(x)T(y)/T(1)$, which result would have been obtained immediately had the strict equality $T(xy) = R(x)S(y)$ been assumed instead of the more

complicated set of equations given in (4.5). Now x may be expressed in the same basis in which the product is obtained, and y must be stored in a basis for which the transformation

$$S(y) = T(y)/T(1) = y^{2^k} \tag{4.17}$$

can be implemented efficiently. There are two simple cases for which this holds: namely $k = 0$, which corresponds to the canonical dual basis; and $\alpha = 1$, so that $T(1) = 1$, which corresponds to Omura's normal basis when $k = 1$.

4.7 The First Bit of the Product

The previous sections have shown that it is always possible to represent the field elements such that the transformations involved in bit-serial multiplication consist of linear feedback shift operations, and that one factor can easily be expressed in the same basis as the product. Up to this point, however, we have ignored the issue of the logic needed to implement equation (4.4), which yields the first bit of the result.

Given that x is expressed in the natural T basis, $\mathrm{B} = \{\mu_0, \mu_1, \ldots, \mu_{m-1}\}$, what logic is involved in producing $\mathrm{Tr}(\alpha_0 xy)$? Let y be represented in an arbitrary basis Q,

$$y = \sum_{j=0}^{m-1} y_j \sigma_j.$$

Then we find

$$z_0 = \mathrm{Tr}(\alpha_0 xy) = \mathrm{Tr}\left(\alpha_0 \sum_{i,j=0}^{m-1} x_i y_j \mu_i \sigma_j\right) = \sum_{i,j=0}^{m-1} x_i y_j \mathrm{Tr}(\alpha_0 \mu_i \sigma_j) = \sum_{i,j=0}^{m-1} x_i y_j A_{ij}, \tag{4.18}$$

where the A matrix depends only on the two bases. It is clear that A is non-singular, for if not there exists a non-trivial linear combination of the σ_j's such that for all $x \in \mathrm{GF}(2^m)$,

$$\mathrm{Tr}\left(\alpha_0 x \sum_{j=0}^{m-1} b_j \sigma_j\right) = 0.$$

In particular, there must be at least one nonzero entry in each row and column of A.

According to (4.18), the logic required to produce the bits of the product involves a GF(2) multiplication (logical AND) operation for each nonzero entry A_{ij}, and the parity over all such pairs. Clearly we would like to choose $\mathrm{Q} = \{\sigma_j\}$ to make A as sparse as possible, as well as to have some sort of regular structure within A so that wiring costs do not dominate. By the above discussion, there must be at least m AND operations with associated parity generation; this minimum will be reached if and only if A is a permutation matrix; i.e., $A_{ij} = \delta_{i,\pi(j)}$, where π is a permutation of $\{0, 1, \ldots, m-1\}$. Not surprisingly, this condition is exactly the prerequisite for Q and B to be dual bases, up to the constant α_0; in other words, $\sigma_j = \alpha_j/\alpha_0$.

Obviously then, we would like to express y in this basis. The question now becomes: can the transformation (4.17) be efficiently applied to the basis Q? In the case of the canonical dual basis (i.e., $k = 0$), where $S(y) = y$ is the identity transformation, any basis will suffice; in particular, we may choose the optimal basis defined by Q. However, when $k \neq 0$, applying

S amounts to a (non-trivial) conjugation, so the choice of bases will be considerably restricted. Consider the action of S on elements of Q. For $j = 0$, $\sigma_0 = 1$, and

$$S(\sigma_0) = S(1) = 1 = \sigma_0.$$

But, for $j = 1, 2, \ldots, m-1$, by (4.13) we find

$$\begin{aligned}
S(\sigma_j) &= S(\alpha_0^{-1}\alpha_j) \\
&= S\left(\alpha_0^{-1}\left(\alpha_0\alpha^{(2^{jk}-1)/(2^k-1)}\right)^{2^{k(m-j)}}\right) \\
&= \alpha_0^{-2^k}\left(\alpha_0\alpha^{(2^{jk}-1)/(2^k-1)}\right)^{2^{k(m-j+1)}} \\
&= \alpha_0^{-2^k}\left(\alpha_0\alpha^{(2^{(j-1)k}-1)/(2^k-1)}\alpha^{(2^{jk}-2^{(j-1)k})/(2^k-1)}\right)^{2^{k(m-j+1)}} \\
&= \alpha_0^{-2^k}\alpha_{j-1}\left(\alpha^{2^{(j-1)k}(2^k-1)/(2^k-1)}\right)^{2^{k(m-j+1)}} \\
&= \alpha_0^{-2^k}\alpha_{j-1}\left(\alpha^{2^{(j-1)k}}\right)^{2^{k(m-j+1)}} \\
&= \alpha_0^{-(2^k-1)}(\alpha_{j-1}/\alpha_0)\alpha^{2^{mk}} \\
&= \sigma_{j-1}\,\alpha\,\alpha_0^{-(2^k-1)}.
\end{aligned}$$

If this operation is to be a simple shift (with or without feedback), we must have $\alpha = \alpha_0^{2^k-1}$, so that example 4.2 applies, with $\omega = \alpha_0$. In this case, however, by (4.14), Q is not a basis, since all its elements are identical. Perhaps there does exist a basis Q where S is not too complicated, but obviously such instances are considerably more complex to wire than the $k = 0$ case. So, when $k \neq 0$, it is not possible both to minimize the $\mathrm{Tr}(\alpha_0 xy)$ logic and to keep the S transformation a simple shift operation.

In particular, a normal-basis representation is inherently not as efficient as the canonical dual basis. For if y is expressed in the basis $\{\sigma_j\}$, we have just seen that the operation of S is difficult to implement. Yet if y is expressed in a normal basis so that S is a simple rotation, it can easily be shown (see Appendix A) that the number of product terms is at least $2m - 1$ and that wiring costs will dominate (because the bases are not dual), forcing the area to grow roughly as m^2. Figure 4-3 shows the wiring required in a simple case, and this complexity will increase for fields of interest in coding.

To summarize, we have now shown that Berlekamp's representation for bit-serial multiplication is in fact the only possible method which requires the minimum circuitry for implementation, and it has the added advantage that multiplication by a fixed constant can be hard-wired. One factor is expressed in a canonical basis, and the other factor is represented in a constant multiple of the corresponding dual basis. The product bits are generated by AND-ing the factors component-wise and taking the parity of the result, then repeatedly applying a simple linear feedback shift to one of the factors in order to output the remaining bits of the product sequentially (see Figure 4-2). Note that, although the parity could be computed using a tree structure, such an approach is not as regular as a linear chain; using steering logic in MOS [41], the delay through a parity chain (for the m of interest) is comparable to the delay through a conventional parity tree. Such a structure can be efficiently implemented in VLSI.

4.8 Self-Dual Bases

At this point let us pose a somewhat more ambitious question. Is it possible for the two bases used in multiplication to be identical, so that there is only one basis needed throughout the decoder system? If such a representation could be found, there would never be the need to perform a basis change, which would certainly simplify the design of a chip. More formally, we would like to know whether there exists a canonical dual basis $\mathrm{B} = \{\mu_i\}$ such that

$$\mathrm{Tr}(\alpha_0 \mu_i \mu_j) = \delta_{i,\pi(j)}, \tag{4.19}$$

where π is some permutation of $\{0, 1, \ldots, m-1\}$. We will call such a basis *self-dual*, since using a single basis the minimum number of product terms is achieved in the bit-serial multiplication logic. Such bases can be characterized in a surprisingly simple way.

Let the canonical basis be $\mathrm{D} = \{\alpha^i \mid i = 0, 1, \ldots, m-1\}$, and define $\alpha_i = \alpha_0 \alpha^i$. Let $\mathrm{B} = \{\mu_i\}$ be the basis dual to $\{\alpha_i\}$. Now note that (4.19) implies that, as sets, $\{\alpha_0 \mu_j\} = \{\alpha_i\} = \{\alpha_0 \alpha^i\}$, or, in other words, $\{\mu_i\} = \{\alpha^i\}$. So equation (4.19) can be satisfied if and only if there exist α, $\alpha_0 \in \mathrm{GF}(2^m)$ such that

$$\mathrm{Tr}(\alpha_0 \alpha^{i+j}) = \delta_{i,\sigma(j)}, \tag{4.20}$$

for $0 \le i, j \le m-1$, where σ is some permutation of the indices. Equation (4.20) has an interesting interpretation in terms of the trace. Suppose we make a list of the trace of consecutive powers of α. Somewhere in this list there must then be m consecutive windows of m bits, each of which contains exactly one nonzero bit.

It will turn out that the form of the minimal polynomial of α determines whether (4.20) can be satisfied, so let $g(x)$ be the minimal polynomial of α,

$$g(x) = \sum_{i=0}^{m} g_i x^i.$$

Obviously $g_m = g_0 = 1$. Since $g(x)$ is irreducible, there must be an odd number of additional nonzero terms g_i, for $1 < i < m$. Define i_l to be the lowest such index, and i_h to be the highest such index. That is, $i_l > 0$ and $g_{i_l} = 1$, but $g_j = 0$ if $0 < j < i_l$. Similarly, $i_h < m$ and $g_{i_h} = 1$, but $g_j = 0$ if $m > j > i_h$. In other words, with exponents arranged in descending order, $g(x) = x^m + x^{i_h} + \cdots + x^{i_l} + 1$; observe that $i_l = i_h$ if and only if $g(x)$ is a trinomial.

Now define λ_j such that $\mathrm{Tr}(\lambda_j \alpha^i) = \delta_{i,j}$, and note that the basis $\mathrm{P} = \{\lambda_j\} = \{\alpha_0 \mu_j\}$ is in fact dual to D itself. An element expressed in the basis P has the form

$$z = \sum_{i=0}^{m-1} z_i \lambda_i = \sum_{i=0}^{m-1} \mathrm{Tr}(z\alpha^i)\lambda_i.$$

In this light, equation (4.20) takes on a different meaning. Namely, the representation of each element of the set $\{\alpha_0 \alpha^j\}$, expressed in the basis P, must have exactly one nonzero bit. Restating this fact in set notation,

$$\{\alpha_0 \alpha^j\} = \{\lambda_i\}. \tag{4.21}$$

So, in particular, for $j = 0$, (4.21) implies that $\alpha_0 \in \mathrm{P}$. Then there exists a k, with $0 < k \le m$, such that $\alpha_0 = \lambda_{k-1}$, and iteratively multiplying λ_{k-1} by α must produce the remaining elements

of P, in some permuted order. But note that multiplication by α of an element expressed in the P basis is a shift operation, just as explained in example 4.1. That is,

$$y = \alpha z = \sum_{i=0}^{m-1} y_i \lambda_i = \sum_{i=0}^{m-1} \lambda_i \mathrm{Tr}(z\alpha^{i+1}) = \sum_{i=0}^{m-2} \lambda_i z_{i+1} + \lambda_{m-1} \mathrm{Tr}(z\alpha^m). \tag{4.22}$$

Now, since α satisfies $g(x)$, (4.22) can be further simplified:

$$y = \alpha z = \sum_{i=0}^{m-2} \lambda_i z_{i+1} + \lambda_{m-1} \sum_{i=0}^{m-1} g_i \mathrm{Tr}(z\alpha^i) = \sum_{i=0}^{m-2} \lambda_i z_{i+1} + \lambda_{m-1} \sum_{i=0}^{m-1} g_i z_i.$$

So, $y_i = z_{i+1}$ for $i = 0, 1, \ldots, m-2$, and $y_{m-1} = \sum_{i=0}^{m-1} z_j g_j$. Clearly $\alpha\lambda_0 = \lambda_{m-1}$, but for $i \neq 0$, $\alpha\lambda_i = \lambda_{i-1}$ if and only if $g_i = 0$. Otherwise, if $g_i \neq 0$, then $\alpha\lambda_i = \lambda_{i-1} + \lambda_{m-1}$, so there are two nonzero bits in the representation of $\alpha\lambda_i$ in this case.

Now suppose that $g(x)$ is not a trinomial. Then, by the above argument, both $\alpha\lambda_{i_l}$ and $\alpha\lambda_{i_h}$ require two bits to be represented in the basis P, implying that there is no way to order all the elements of P such that consecutive entries have a ratio of α. Thus, in this case it is not possible to satisfy (4.20). However, if $g(x)$ is a trinomial, $i_h = i_l = k$, then the ordering

$$\lambda_{k-1}, \lambda_{k-2}, \ldots, \lambda_0, \lambda_{m-1}, \lambda_{m-2}, \ldots, \lambda_{i_k} = \alpha_0, \alpha_0\alpha, \ldots, \alpha_0\alpha^{m-1}$$

is seen to satisfy all requirements. In other words, we pick $\alpha_0 = \lambda_{k-1}$, so that $\lambda_k = \alpha_0\alpha^{m-1}$ is the last element in the power sequence. We have just proven the following.

Theorem 4.8. *Equation (4.20) has a solution if and only if α satisfies an irreducible trinomial $g(x)$ over GF(2). If $g(x) = x^m + x^k + 1$, then choose $\alpha_0 \neq 0$ such that $\mathrm{Tr}(\alpha_0\alpha^j) = 0$ for all $j \neq k-1$, $0 \leq j \leq m-1$.*

Proof: By the above argument, $g(x)$ must be a trinomial, and we have shown by construction that if it is a trinomial, $\alpha_0 = \lambda_{k-1}$ will satisfy (4.20). The definition for α_0 given in the statement of the theorem is identical to the definition of λ_{k-1}. ∎

A few comments are in order at this point, followed by some examples. First, the minimal polynomial does not have to be primitive for the above construction to work, although a primitive root may simplify other aspects of the decoding procedure. Also, by Swann's theorem [4], there are no irreducible trinomials of degree m when m is a multiple of eight. Therefore, in particular it is not possible to find a self-dual basis over GF(256), which is unfortunate because this field has been widely used in Reed-Solomon coding systems, such as the NASA standard code [20] and compact audio disks, and because eight-bit quantities are a common data size. An examination of a table of irreducible trinomials (see Appendix C) reveals however that such trinomials exist for all other $m < 16$, except $m = 13$. If the basis used in multiplication is not self-dual, the penalty may not be too great, because, for most of the decoder architectures we will consider in Chapter six, basis transformations are required not at each multiplier but at only a few places on the chip. Also, we will give an example of the procedure for minimizing the cost of a basis change below. However, since for the fields of interest other than GF(256) a self-dual basis is available, such a basis should be selected for VLSI decoder implementation over these fields if a dual-basis structure is required.

Example 4.3: Self-Dual Basis over GF(128)

The polynomial $g(x) = x^7 + x + 1$ is primitive, so let α be a root. Using the canonical basis for GF(128) generated by α, with $k = 1$ in the above theorem, we want to choose α_0 such that $\mathrm{Tr}(\alpha_0 \alpha^i) = 0$ for $0 < i \leq 6$. It turns out that over this field, $\mathrm{Tr}(\alpha^i) = 0$ for $0 < i \leq 6$, so $\alpha_0 = 1$. Then, $\mathrm{Tr}(1) = 1$ implies that $\mu_0 = 1$, and for $1 \leq i \leq 6$, $\mathrm{Tr}(\alpha^i) = 0$ means that $\mu_i = \alpha^{7-i}$, since clearly $\mathrm{Tr}(\alpha^7) = 1$. Here is a list of the canonical basis elements with the corresponding elements of the dual basis B.

$$\begin{array}{ccccccc} 1 & \alpha & \alpha^2 & \alpha^3 & \alpha^4 & \alpha^5 & \alpha^6 \\ 1 & \alpha^6 & \alpha^5 & \alpha^4 & \alpha^3 & \alpha^2 & \alpha . \end{array}$$

Note that the elements in the dual basis are in reversed order from the canonical basis after the first element. It can readily be seen that this will be the case in general when $x^m + x + 1$ is the minimal polynomial, and perhaps this fact could be used in building the bit-serial multiplier by laying out the two registers holding x and y in opposite order. ∎

Example 4.4: Self-Dual Basis over GF(64)

The polynomial $g(x) = x^6 + x^3 + 1$ is irreducible, and any root α of $g(x)$ has order 9. Obviously, $\mathrm{Tr}(1) = 0$ in this field, and $g_5 = 0 = \mathrm{Tr}(\alpha)$ implies

$$\mathrm{Tr}(\alpha) = \mathrm{Tr}(\alpha^2) = \mathrm{Tr}(\alpha^4) = 0.$$

Also, note that $(\alpha^5)^2 = \alpha$, so $\mathrm{Tr}(\alpha^5) = 0$. Because at least one element of any basis has nonzero trace, we must have $\mathrm{Tr}(\alpha^3) = 1$, which can indeed be verified. Observe that since $g(x)$ is reversible, α^{-1} is also a root, so

$$\mathrm{Tr}(\alpha^{-1}) = \mathrm{Tr}(\alpha^{-2}) = \mathrm{Tr}(\alpha^{-4}) = 0.$$

Similarly we find $\mathrm{Tr}(\alpha^{-3}) = 1$ and $\mathrm{Tr}(\alpha^{-5}) = 0$. Using these facts, we see from the theorem that $\alpha_0 = \alpha^{-5}$. Here is the pairing of dual elements.

$$\begin{array}{cccccc} 1 & \alpha & \alpha^2 & \alpha^3 & \alpha^4 & \alpha^5 \\ \alpha^2 & \alpha & 1 & \alpha^5 & \alpha^4 & \alpha^3 . \end{array}$$

As in the previous example, note that there are subsets of the basis for which the dual elements appear in reverse order. Here α is not primitive, which could make implementation of a Chien search more difficult, because continued multiplication by α does not generate all field elements. Perhaps for this field the primitive polynomial $x^6 + x + 1$ would be a better choice; however, it is clear that the theorem can be applied in either case. ∎

After studying these examples, it becomes evident that the dual basis pairings can be determined without going into involved trace calculations. In general, if $x^m + x^k + 1$ is irreducible, the dual basis ordering is as follows:

$$\begin{array}{ccccccc} 1 & \alpha & \ldots & \alpha^{k-1} & \alpha^k & \ldots & \alpha^{m-1} \\ \alpha^{k-1} & \alpha^{k-2} & \ldots & 1 & \alpha^{m-1} & \ldots & \alpha^k . \end{array}$$

That is, for $0 \leq i < k$, $\mu_i = \alpha^{k-1-i}$, and for $k \leq i < m$, $\mu_i = \alpha^{m-1-(i-k)}$. Finding the value of α_0 will usually involve a search through the traces of field elements; however, it is not necessary to know the value of α_0 to implement the multiplier. This remarkably simple result allows us to determine the dual basis immediately upon inspection of the minimal trinomial. To our knowledge, no one has previously derived such a result.

4.9 Nearly Self-Dual Bases

Because there are no irreducible trinomials of degree eight, we have no hope of finding a self-dual basis for GF(256). However, using the insight gained from the previous discussion, a basis which is almost self-dual can be obtained. Before outlining the construction of such a basis, let us derive a bound on how close a basis can be to self-dual in such instances. In fact, we shall see that this bound is tight for the cases of interest.

Corollary 4.9. *If no irreducible trinomial of degree m exists, any scalar multiple of the basis dual to a given canonical basis D for $\mathrm{GF}(2^m)$ must differ from D in at least two elements.*

Proof: To see this, suppose instead that, for $j = 0, 1, \ldots, m-1$,

$$\mathrm{Tr}(\alpha_0 \alpha^{i+j}) = \begin{cases} \delta_{i,\sigma(j)}, & j \neq j_0; \\ \sum_{k=0}^{m-1} b_k \delta_{i,k}, & j = j_0 \end{cases}, \tag{4.23}$$

where $b_k \neq 0$ for at least two values of k. Equation (4.23) implies that the basis $\mathrm{P} = \{\lambda_j\} = \{\alpha_0 \mu_j\}$ contains all but one element of the sequence $\alpha_0 \alpha^j$ of m consecutive powers of α. Let $\alpha_0 \alpha^k$ be the only element of the sequence which is not in P, and again consider the multiplication of a field element, expressed in the basis P, by α. There are now two cases of interest.

If $k = 0$ or $k = m-1$, then clearly there exists an element λ_l such that $\lambda_l \alpha^j \in \mathrm{P}$ for $j = 0, 1, \ldots, m-2$. However, from the argument preceding Theorem 4.8, it is clear that this condition can occur only if $i_h - i_l < 2$, which is impossible since the difference $i_h - i_l > 1$ for $g(x)$ not a trinomial, because any irreducible polynomial over GF(2) must have an odd number of nonzero coefficients.

On the other hand, suppose $0 < k < m-1$. Then both $\alpha_0 \alpha^{k-1}$ and $\alpha_0 \alpha^{k+1}$ are elements of P, implying that their representations in this basis have exactly one nonzero component. However, $\alpha_0 \alpha^k$ is not in the basis, so its representation must have more than one nonzero bit, and since it is a (linear feedback) shift of $\alpha_0 \alpha^{k-1}$, there must be exactly two nonzero bits. In other words, $\alpha_0 \alpha^k = \lambda_i + \lambda_{m-1}$ for some i. Applying one more shift to the register must yield only a single nonzero bit for $\alpha_0 \alpha^{k+1}$, which will occur only if $i = 0$ and $g_{m-1} = 1$, which further implies that $\alpha_0 \alpha^{k-1} = \lambda_1$, $\alpha_0 \alpha^{k+1} = \lambda_{m-2}$, and $g_1 = 1$. Again, there must be at least one additional nonzero coefficient of $g(x)$, call it g_a, with $1 < a < m-1$. Then, the sequence of powers can contain at most the following elements of P

$$\lambda_{a-1}, \lambda_{a-2}, \ldots, \lambda_1, \lambda_{m-2}, \lambda_{m-3}, \ldots, \lambda_a, \tag{4.24}$$

because otherwise the representation of some other element of the sequence will have two nonzero bits. Now observe that in (4.24) there are only $a-1+m-1-a = m-2$ elements, but we needed $m-1$ such elements to satisfy (4.23). So, by contradiction, if there is no irreducible trinomial, there must be at least two elements of the dual basis which are not in the canonical basis. ∎

Now the question arises: is it possible to find such a basis when there are no irreducible trinomials? From the arguments leading up to Theorem 4.8, the key observation is that to minimize the cost of a basis change, a minimal polynomial for which i_l and i_h differ by as little as possible should be selected. As noted above, the difference $i_h - i_l$ for non-trinomials must be at least two, and this minimum occurs only if $g(x)$ is a so-called *pentanomial* with the form

$$g(x) = x^m + x^{k+1} + x^k + x^{k-1} + 1. \tag{4.25}$$

Proceeding with an analysis similar to that of the preceding theorem, we find the only differences occurring at the boundary conditions; namely, the first and last elements of the sequence of powers of α. The dual basis pairings are as follows:

$$\begin{array}{cccccccc} 1 & \alpha & \cdots & \alpha^{k-2} & \alpha^{k-1} & \alpha^{k} & \alpha^{k+1} & \cdots & \alpha^{m-1} \\ \alpha^{k-1} & \alpha^{k-2} & \cdots & \alpha & 1+\alpha^{k-1} & \alpha^{m-1}+\alpha^{k} & \alpha^{m-2} & \cdots & \alpha^{k}. \end{array}$$

Obviously, the bases differ in only two elements, and only two XOR operations are required to perform a basis change.

The next question that arises is: when do irreducible pentanomials of the form given by (4.25) exist? Obviously, we are only interested in the cases where no trinomials are available, namely $m = 8, 13, 16, 19, 24, 26, 27, 32$, etc. For $m = 8$, it is not too difficult to find such a polynomial by inspection of a table [44]. However, for larger values of m, a computer search is appropriate. Such a program has been written and has found the following primitive polynomials for $m \leq 32$:

$$\begin{array}{ccccccccc} x^{8} & + & x^{4} & + & x^{3} & + & x^{2} & + & 1 \\ x^{13} & + & x^{7} & + & x^{6} & + & x^{5} & + & 1 \\ x^{16} & + & x^{5} & + & x^{4} & + & x^{3} & + & 1 \\ x^{19} & + & x^{7} & + & x^{6} & + & x^{5} & + & 1 \\ x^{26} & + & x^{7} & + & x^{6} & + & x^{5} & + & 1 \\ x^{32} & + & x^{13} & + & x^{12} & + & x^{11} & + & 1. \end{array}$$

Obviously, the reciprocal polynomials are also irreducible, and there are no non-primitive irreducible polynomials of this form over these fields. For $m = 24$ and $m = 27$ there are no such irreducible pentanomials, but it is clear that, for all fields of interest, either a self-dual basis or a nearly self-dual basis is available.

Example 4.5: Nearly Self-Dual Basis over GF(256)

Let α be a root of the primitive polynomial

$$g(x) = x^8 + x^4 + x^3 + x^2 + 1.$$

Here $k = 3$. After some searching we find the dual basis pairings are

$$\begin{array}{cccccccc} 1 & \alpha & \alpha^{2} & \alpha^{3} & \alpha^{4} & \alpha^{5} & \alpha^{6} & \alpha^{7} \\ \alpha^{-3} & \alpha^{-4} & \alpha^{45} & \alpha^{98} & \alpha & 1 & \alpha^{-1} & \alpha^{-2}. \end{array}$$

Choosing $\alpha_0 = \alpha^{-5}$, the correspondence looks much more promising:

$$\begin{array}{cccccccc} 1 & \alpha & \alpha^{2} & \alpha^{3} & \alpha^{4} & \alpha^{5} & \alpha^{6} & \alpha^{7} \\ \alpha^{2} & \alpha & \alpha^{50} & \alpha^{103} & \alpha^{6} & \alpha^{5} & \alpha^{4} & \alpha^{3}. \end{array}$$

Now the only two elements of the dual basis which are not also found in the canonical basis are

$$\alpha^{50} = \alpha^2 + 1$$
$$\alpha^{103} = \alpha^3 + \alpha^7.$$

In other words, only two XOR gates and some wiring are required to change from the dual basis to the canonical basis, as illustrated in Figure 4-4. Note that the total gate count in such a case (including the multiplier and the basis change) is still less than that required for a normal basis multiplier over the same field, and the wiring is considerably simpler. ∎

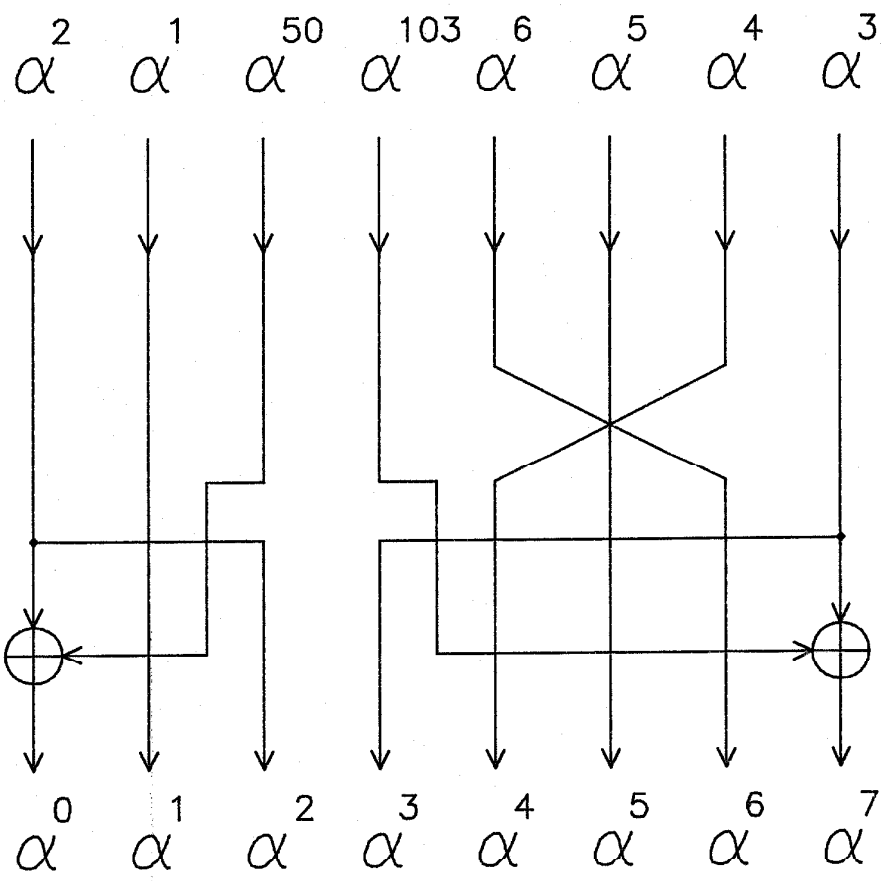

Figure 4-4. Dual to Canonical Basis Change Over GF(256)

4.10 Shift-and-Add Multipliers

Let us attempt to generalize the multiplication method presented in Figure 4-1. We will be somewhat less rigorous in our derivation here, but it is clear that the methods of the previous sections could be applied to formalize these ideas. As shown in Figure 4-5, the basic concept is that the partial products $x_i y$, which each involve only m multiplications over GF(2), are produced sequentially and added to a transformed version of the current partial sum to generate the next partial sum. The function R which is applied to the register is a linear transformation which we hope to express in hardware as a simple feedback shift register. Once all the bits of x have been input, the register should hold the product $z = xy$, so all of the result bits will become available in parallel.

Given a basis $\mathrm{B} = \{\mu_0, \mu_1, \ldots, \mu_{m-1}\}$, without loss of generality we may assume that the bits of x are presented in the order $x_{m-1}, \ldots, x_1, x_0$. If we define $z^{(0)} = 0$, successive partial sums can be computed using

$$z^{(k)} = R\Big(z^{(k-1)}\Big) + x_{m-k} y \quad \text{for} \quad k = 1, 2, \ldots, m\,.$$

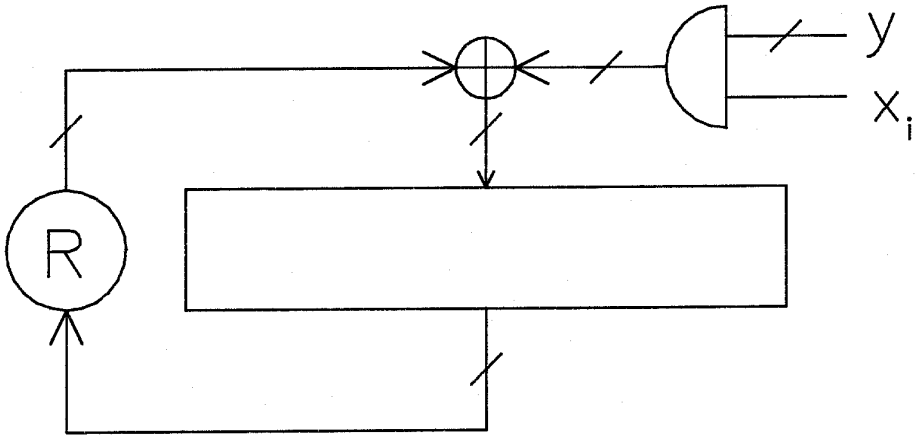

Figure 4-5. General Shift-and-Add Multiplier

After m iterations, the result $z^{(m)} = xy$ is obtained. Since R is linear, the expression for the product can be simplified:

$$z = z^{(m)} = xy = \sum_{i=0}^{m-1} x_i \mu_i y = \sum_{i=0}^{m-1} R^i(x_i y) = \sum_{i=0}^{m-1} x_i R^i(y). \tag{4.26}$$

Now notice that (4.26) implies that $R^i(y) = \mu_i y$ for all y. In particular, for $i = 0$ we find $\mu_0 = 1$, and the case $i = 1$ implies that $R(y) = \mu_1 y$. Then, by induction, $\mu_i = \alpha^i$ for some $\alpha \neq 0$; in other words, B is a canonical basis.

Thus, the shift-and-add technique is fairly simple. The factor x must be expressed in a canonical basis, but, because the holding register must be capable of multiplying itself by α, both y and the product z should be expressed in a basis which facilitates this transformation: the dual basis is possible, but a canonical basis may also be desirable. In the latter case, the largest XOR structure involves only three bits, as in Figure 4-1, eliminating the need for a large parity tree altogether, which should be very attractive for performance reasons. Observe that, since multiplication of x by a constant can be hard-wired, using this technique it is possible to design a bit-serial Reed-Solomon encoder similar to (although somewhat less efficient than) that presented in [5]. Clearly the shift-and-add structure has many of the advantages of the dual-basis multiplier; the superiority of one approach over the other will depend on the particular problem and technology of interest.

4.11 Other Bit-Serial Operations

Our search for efficient bit-serial multiplier structures has been quite successful. However, the decoding algorithms also involve two other arithmetic operations over finite fields: addition, and multiplicative inversion. Using any basis, addition can be done bit-serially using an XOR gate, but taking the inverse of an element is of much greater concern. There are versions of each decoding algorithm which avoid inversion during the calculation of the error polynomials. Some authors and designers [10,31] have also advocated the use of recursive extension in the

frequency domain to complete the error correction without resorting to the Forney algorithm (which requires taking a reciprocal). While this approach is conceptually enlightening, casting such techniques into hardware involves structures with $2^m - 1$ elements, with each element consisting of a multiplier-accumulator, and performing a sum over the outputs of all the elements. This approach is markedly more expensive in chip area than the use of a simple lookup table to accomplish inversion.

It is perhaps not surprising that the same basis which is used to optimize bit-serial multiplication lends itself naturally to bit-serial inversion. Suppose that we have a table (ROM) with 2^m single bit entries, which implements the function $f(x) = \mathrm{Tr}(\alpha_0 x^{-1})$, for $x \neq 0$. Clearly this result is the first bit of x^{-1}, expressed in the dual basis. Now the next bit of the reciprocal is $\mathrm{Tr}(\alpha_0 \alpha x^{-1}) = f(x\alpha^{-1})$. Similarly, the remaining bits of the x^{-1} are produced by dividing x by α repeatedly and looking up the corresponding bit in the table. Division by α is also a simple shift operation in the dual basis, but the shift is in the opposite direction from that involved in multiplication, and the feedback term is to μ_0 instead of μ_{m-1}. This method of bit-serial inversion can also be adapted to work with normal bases, by squaring x each time. When using a canonical basis and shift-and-add multipliers, bits of the reciprocal cannot be produced from a lookup table in a totally independent fashion, but with the addition of a simple linear feedback shift register it is still possible to accomplish bit-serial inversion. For the field sizes of interest, say $m \leq 12$, such a ROM will easily fit onto a small part of the chip.

Another operation which is simple to implement bit-serially is that of taking the square root. Given a register holding a value x expressed in the canonical dual basis, suppose we wish to find y such that $y^2 = x$. Note that $y = x^{2^{m-1}}$. We implement a parity circuit which produces $y_0 = \mathrm{Tr}(\alpha_0 y)$. The next bit of y is

$$y_1 = \mathrm{Tr}(\alpha_0 \alpha y) = \mathrm{Tr}\Big(\alpha_0 (\alpha^2 x)^{2^{m-1}}\Big).$$

In other words, we multiply x by α^2 to produce successive bits of the square root. If the register is wired to perform multiplication by α, two clocks are required to produce each bit of y. Another suggestion, due to McEliece, is to perform a basis change on x to the canonical dual basis corresponding to α^2. Since α and α^2 satisfy the same minimal polynomial, the register for x is identical (although the parity for y_0 is different), but now each clock produces one bit of the square root. Similar results can be obtained for taking other conjugates in a canonical dual basis, although it is clear that a normal basis is best suited for such operations. Extensions to canonical bases can be made, but are somewhat more involved, as in the case of inversion.

In fact, conjugation and inversion are but two examples of a more general operation, namely exponentiation. If the exponent is a power of two, the operation is linear and the lookup table consists of a parity circuit. However, for other exponents, a full lookup table is needed, as in the case of taking a reciprocal.

Thus we see that the arithmetic operations involved in decoding algorithms can be quite efficiently performed in a bit-serial fashion, all involving a single basis. The selection between structures using the dual basis and the canonical basis may be based on the particular implementation technology. However, if a self-dual basis is available, changing bases has a very low cost, so the structure most appropriate for the timing needs of the problem at hand can be selected at various points in the architecture. Using these techniques, a decoder chip can be designed in which all global communication is bit-serial, greatly reducing the area and power needed for a given performance level.

Chapter 5

Reed-Solomon Decoding Algorithms

5.1 Historical Overview

Perhaps the single most important development in the theory of block codes is the discovery of efficient algebraic decoding techniques for BCH and Reed-Solomon codes. For a code of a given minimum distance, these algorithms can be executed in polynomial time on a sequential machine; thus they have some practical advantages over decoding schemes for convolutional codes, which are generally exponential in either time or space [40, Ch. 9,11]. In this chapter the known Reed-Solomon decoding algorithms are presented, and their features are explained with an eye toward VLSI implementation. By displaying the algorithms side-by-side in a common format, it is hoped that the many similarities between these techniques can be observed and clearly understood.

The original descriptions of BCH codes were published in 1959 [30] and 1960 [12], and the discovery of Reed-Solomon codes came very soon thereafter [45]. However, these papers briefly explained how to construct codes having the desired distance properties, without providing an accompanying decoding procedure. Within a year Peterson, [43] devised a decoding scheme for BCH codes which required the solution of a system of linear equations involving the power-sum syndromes over the finite field. Although this technique was later extended to include Reed-Solomon codes [28] and simplifications were added [17,26], such an approach is inherently at least cubic in the number of errors which occurred, and almost a decade passed before more efficient decoding algorithms were devised.

In 1968, Berlekamp published a book, *Algebraic Coding Theory* [4], which included a chapter detailing an iterative decoding algorithm for solving what he termed the *key equation*, a simplified formulation of Peterson's matrix equations. Massey [37] later interpreted Berlekamp's method as a solution for the general problem of synthesizing the shortest linear feedback shift register capable of generating a given finite sequence of values; in particular, the power-sum syndromes are the sequence of interest in this case. The algorithm has roughly quadratic complexity in terms of the redundancy and has a fairly regular structure. Berlekamp applied the procedure to both Reed-Solomon and BCH codes, handling erasures as well as errors. With great foresight he also showed how to take advantage of the parallelism in this algorithm by using a number of identical slave processors and one master controller [4, Section 7.7], in a fashion very similar to the architectures presented in Chapter six. With the advent of Berlekamp's key

equation solver, it finally became possible to build fairly simple and efficient decoders.

Unfortunately, Berlekamp's algorithm, although elegant and simple, can appear somewhat unmotivated to the novice. In 1975, it was discovered that Euclid's algorithm [47] could be used for decoding, by interpreting the key equation as a problem in rational approximation. This method cast the decoding process onto more familiar mathematical ground, providing an analytical motivation for the algorithm. Nonetheless, there are many similarities between the two algorithms, and recently Cheng [16] has proved their equivalence. Euclid's algorithm has generally been regarded as being somewhat less efficient for implementation purposes [10,19,40], which may be true for sequential machines; however, Kung [14] has shown that it can be implemented with a systolic array, presenting the possibility of high-performance pipelined decoders in VLSI.

Another major conceptual breakthrough occurred in the late 1970's, when it was realized that cyclic error-correcting codes could be described in terms of well-known digital signal processing techniques, such as the discrete Fourier transform and spectral estimation [9, 42]. In fact, this viewpoint is very close in spirit to the original presentation of Reed and Solomon [45]. Codewords are considered to be time-domain vectors which satisfy certain constraints in the frequency domain; namely, the syndromes, which are but contiguous elements of a discrete Fourier transform (spectrum) over the field, must all be zero. Encoding can be accomplished starting in the frequency domain with a vector which satisfies the spectral constraints and performing an inverse transform to generate a time-domain codeword, although such an encoder is non-systematic and is not practically efficient. Given a time-domain word, decoding involves a forward transform to obtain a window of the error spectrum (where the codeword spectrum is zero), estimating the entire error spectrum from this window, and performing an inverse transform to produce the time-domain error pattern. Such terminology is much more familiar to the engineer concerned with actual implementation of error control systems. There are many variations on this theme; for example, the final inverse transform may be simplified using a Chien search, or the initial transform may be performed at either the encoder or the decoder. The interested reader is encouraged to refer to Blahut [10, Chapters 8,9] for a lucid explanation of these concepts.

Noticing that all the decoding systems involved both a forward and an inverse transform on the data at some point, Blahut [10, Section 9.5] realized that the transform could be applied directly to the algorithm instead of the data. His time-domain decoder is a version of the Berlekamp-Massey algorithm, transformed into the time domain. Unfortunately, for a t error-correcting Reed-Solomon code, such a transformation extends vector quantities of size $2t$ in the frequency domain to size n in the time domain, so variations of this algorithm have area-time complexity of nt or n^2. Thus, although the structure of such a decoder is quite simple, it seems to be limited to low-speed and/or low-rate applications. In particular, for such a decoder to run in full synchrony with the incoming data, the hardware would require area proportional to n, which is prohibitively expensive, especially for high-rate codes.

Calculation of the power-sum syndromes has always been a major bottleneck in conventional decoding systems [19]. With Berlekamp's discovery of extremely efficient encoders for Reed-Solomon codes over GF(2^m), the question naturally arose whether there was a decoding method which utilized the remainder polynomial directly, since the remainder contains exactly the same information as the syndromes, as we saw in Chapter three. If such algorithms could be found, the received word could be re-encoded in hardware to produce the remainder, thus bypassing one of the most costly aspects of the decoding procedure. Finally, this milestone was reached in 1982 when Berlekamp and Welch succeeded in deriving an algorithm [8], similar in flavor to previous decoding procedures, involving only the use of the remainder polynomial.

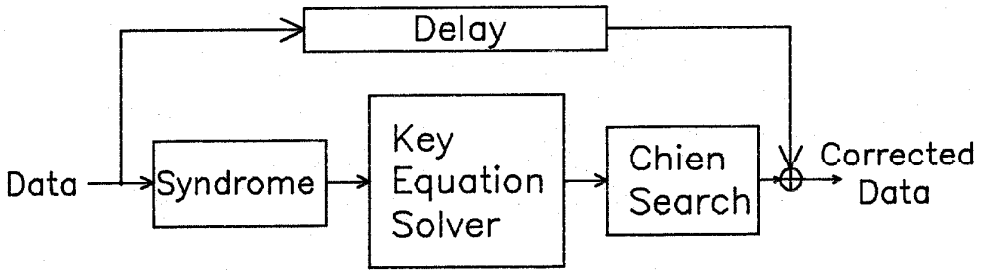

Figure 5-1. Block Diagram of Typical Syndrome Decoder

Again, as with Berlekamp's original key equation solver, the algorithm was beautiful, elegant, and seemingly unmotivated. The following year, one of Welch's students, T. H. Liu, came up with a very similar algorithm [35], but succeeded in relating it formally to the syndrome key equation and in motivating it by analogy with the Berlekamp-Massey algorithm.

In retrospect, it is almost as easy to compute the syndromes in hardware as the remainder, using Berlekamp's bit-serial techniques, so these time-domain decoding schemes may not be as significant for implementation as was originally hoped. However, the newer algorithms afford a greater perspective on decoding: with so many methods to examine, similarities between algorithms can readily be recognized and generalized. Also, modifications which enhance one particular decoding procedure can often be employed under a slightly different guise in other algorithms. It is hoped that the numerous decoding schemes presented in this chapter will allow the reader to gain some intuition about such generalizations.

5.2 The Key Equation Revisited

With the exception of Blahut's time-domain decoder, which has only one main phase, every Reed-Solomon decoding algorithm consists of three distinct steps, as depicted in Figure 5-1. The first stage is the generation of the power-sum syndromes or the remainder polynomial from the received word. Next, the error polynomials (e.g., σ and ω) are derived in a step often known as the key equation solution. Finally, a Chien search (or inverse transform) is used to locate and correct the errors. Of these phases, the first and last do not vary from algorithm to algorithm within a given class of decoders. For example, any syndrome decoding method will operate properly regardless of how the syndromes are generated; similarly, the Chien search proceeds without a knowledge of how the error locator and evaluator were obtained. Thus, although these steps will be included in the description of each algorithm in this chapter, the main emphasis here will be on the solution of the key equation. The following chapter on decoder architecture will investigate in detail the alternatives for implementation of each major step of the procedure.

The key equation has two basic forms. For the power-sum syndromes,

$$\sigma(x)S(x) \equiv \omega(x) \pmod{x^{2t}}, \tag{5.1}$$

where σ is the error-locator polynomial, ω is the error evaluator, and S is the syndrome polynomial, as we saw in Chapter three. In the algorithms to follow, there will be a slight notational inconsistency with our convention of upper-case letters for frequency-domain quantities. Often

we will make the definition $S_k := s(\alpha^{L+k})$, whereas this value should actually be assigned to S_{L+k}; this change is made only for convenience in defining the syndrome polynomial $S(x)$. When using the remainder polynomial, let us say $s(x)$ is the received word, and

$$r(x) = \sum_{i=0}^{2t-1} r_i x^i \equiv s(x) \pmod{g(x)},$$

where $g(x)$ is the generator polynomial for the code. It has then been shown [8,35] that, if there exist polynomials N and W, each of degree less than $t+1$, such that, for $i = 0, 1, \ldots, 2t-1$,

$$c_i N(\alpha^i) = r_i W(\alpha^i), \tag{5.2}$$

where the c_i are nonzero constants depending only on the code, then N and W can be thought of as the error-evaluator and error-locator polynomials, respectively. In particular, the roots of W correspond to error locations; for errors occurring in the information characters, the error values can be evaluated as

$$e_k = f(\alpha^k) N(\alpha^k) / W'(\alpha^k),$$

where f is a function which depends only on the code and could be stored in a ROM. Clearly a Chien search can be applied just as in the syndrome case. However, errors in the parity characters must be corrected by reencoding. The various remainder decoding algorithms involve iterative techniques for solving (5.2).

Many similarities between the algorithms will become apparent as they are presented. In particular, at each iteration of the key equation solution, the current best estimate of the error polynomials is evaluated, and an update occurs to give an improved estimate, based on the result of the evaluation. This evaluate-update cycle is repeated until the key equation is solved. We will see that the evaluation criterion varies from algorithm to algorithm, but the update step always involves a linear combination of the current estimates. The ability of each algorithm to handle erasure decoding will also be discussed; in general, decoding erasures and errors involves a slightly more complex initialization procedure which replaces the first few iterations of the algorithm. Each of the algorithms presented in the following sections (including all the modified versions) has been implemented in a high-level programming language and tested for a variety of codes and input data. Our presentations will be given at a fairly low level; for example, in the Euclidean algorithm, polynomial division will be represented as a series of shifts and adds. Hopefully such an approach will indicate how the operations could actually be implemented in hardware.

Perhaps the best way to illustrate these points is to present one algorithm in detail. For historical purposes we have selected the original Berlekamp key equation solver, and in the following section several modifications of this algorithm will be explained, including a version which can handle erasures. Because many of these modifications are quite general, the other algorithms will not be presented in such detail, but the reader should be able to intuit how similar changes can be applied in any decoder.

5.3 Berlekamp Algorithm

In Figure 5-3, Berlekamp's original key equation solver [4] is presented in pseudocode. The three major loops in the code correspond to the functional blocks of Figure 5-1. Four polynomials are

involved in the solution of the key equation: $\sigma(x)$ is the error locator, $\omega(x)$ is the error evaluator, and $\tau(x)$ and $\gamma(x)$ are the corresponding auxiliary polynomials. The pair (σ,ω) represents the current best estimate of the error polynomials, while the pair (τ,γ) is a linearly independent previous best estimate. A total of $2t$ iterations are required to find the appropriate final values for σ and ω; each time through the loop a new guess is made, and, for example, $\sigma^{(k)}$ represents the estimate of σ after the loop has been executed k times. Because the auxiliary polynomials are linearly independent from the main polynomials, when the evaluation step finds a nonzero discrepancy, a linear combination of the pairs is used to select a new estimate with zero discrepancy. We shall see that this general theme is repeated throughout the decoding algorithms; in fact, some form of every decoding scheme has four polynomials which are used in an identical fashion.

The correctness of the decoder is not at issue here; the interested reader should refer to Berlekamp [4] or some other good coding text for such a proof. Instead, it is our aim to come to an understanding of the motivation behind Berlekamp's algorithm, which interprets the key equation (5.1) as a problem in polynomial multiplication. If $\deg(\sigma) = e$, then note that, since $\deg(\omega) < e$, there are $2t - e$ consecutive zero coefficients of the product $\sigma(x)S(x)$. Each of these coefficients can be viewed as the result of a convolution (or inner product), as in the expression for Δ in the pseudocode; equivalently, from the Massey viewpoint [37], this coefficient is the output of a linear feedback shift register (LFSR) with feedback taps given by σ which is initialized by the low order coefficients of $S(x)$ (see Figure 5-2), assuming that $\sigma(x)$ has been normalized so that the constant coefficient is 1. Given an estimate of σ for which the corresponding LFSR outputs h consecutive zeroes, the evaluation step consists of checking to see whether the next output in the sequence, Δ, called the *discrepancy*, is also zero. If so, the current estimate of σ is good for the next iteration; otherwise a new estimate must be made.

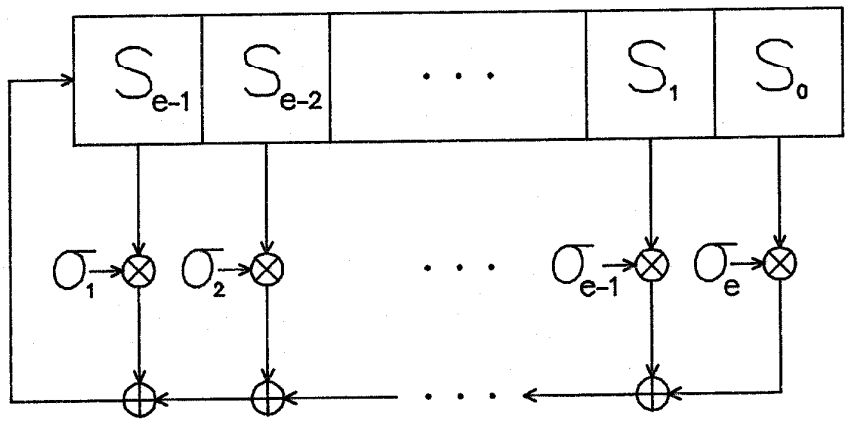

Figure 5-2. Massey's View of Key Equation

The auxiliary polynomial $\tau(x)$ is always maintained so that the corresponding LFSR produces $h-1$ consecutive zeroes followed by 1. Thus the LFSR for $x\tau(x)$ generates h zeroes followed by 1, assuring that the linear combination $\sigma(x) - \Delta x\tau(x)$ has zero discrepancy. Note that $\tau(x)$ is updated either by selecting a scaled version of the old value of $\sigma(x)$ or by multiplication by x, which corresponds to lengthening the LFSR, with the choice based on the desire to minimize the

```
/*          ASSUMPTIONS:                                    */
/* 1. Codewords are multiples of g(x) = ∏_{k=L}^{L+2t-1} (x - α^k) . */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword. */

PROCEDURE  Berlekamp;
BEGIN
                    /* first calculate the syndromes */
   FOR k := 0,1,...,2t - 1 DO S_k := Σ_{i=0}^{n-1} s_i α^{i(k+L)};
                    /* initialize for key equation solver */
   D  :=  0;
   σ^(0)(x)  :=  τ^(0)(x)  :=  ω^(0)(x)  :=  1;
   γ^(0)(x)  :=  0;
   FOR k := 0,1,...,2t - 1 DO    /* solve key equation */
   BEGIN          /* evaluate (i.e., convolve) */
      Δ  :=  Σ_{i=0}^{k} σ_i^(k) S_{k-i};
                    /* update */
      σ^(k+1)  :=  σ^(k) - Δxτ^(k);
      ω^(k+1)  :=  ω^(k) - Δxγ^(k);
      IF Δ = 0 OR (2D > k + 1) THEN
                    τ^(k+1)  :=  xτ^(k);
                    γ^(k+1)  :=  xγ^(k);
      ELSE
                    D  :=  k + 1 - D;
                    τ^(k+1)  :=  Δ^{-1}σ^(k);
                    γ^(k+1)  :=  Δ^{-1}ω^(k);
      ENDIF;
   END;
   σ(x)  :=  σ^(2t)(x);
   ω(x)  :=  ω^(2t)(x);
   FOR k := 0,1,...,n - 1 DO     /* Chien search */
      IF σ(α^{-k}) = 0 THEN s_k := s_k - α^{-k(L-2)}ω(α^{-k})/σ'(α^{-k})    ENDIF;
END;
```

Figure 5-3. Berlekamp Algorithm

degree of τ. In general, at some point the polynomial degrees must enter into the evaluation phase of any decoding algorithm, because our aim is to solve the key equation subject to constraints on these degrees. The variable D in the algorithm is an upper bound on the degree of $\sigma(x)$, and it can easily be shown that

$$\deg\left(\sigma^{(k)}(x)\right) + \deg\left(x\tau^{(k)}(x)\right) \leq k+1 . \tag{5.3}$$

Thus, the choice of $\tau^{(k+1)} = \Delta^{-1}\sigma^{(k)}$ as an update has lower degree if $D < k + 1 - D$, which is exactly the condition tested in the pseudocode. Actually, Berlekamp's initial statement of the procedure was slightly more complicated, with an additional Boolean variable used to insure that equality would hold in some relationships involving the polynomial degrees and the parameter D. However, it is interesting that the algorithm performs correctly without this constraint; in the conditional statement, when $\Delta \neq 0$ and $2D = k + 1$, the choice as to which branch to take is in fact arbitrary.

The error-evaluator polynomial ω and its corresponding auxiliary polynomial γ are initialized differently from σ and τ, but the update phase is applied identically to them throughout the algorithm. It should be observed that the algorithm actually does not compute the formal value of the error evaluator as described in Chapter three, which should have degree strictly less than the error locator. Berlekamp uses a slightly modified (but entirely equivalent) version of the key equation, given by

$$\sigma(x)(1 + xS(x)) \equiv \omega(x) \pmod{x^{2t+1}}. \tag{5.4}$$

Clearly the computed $\omega(x)$ is the sum of the error-locator polynomial and x multiplied by the formal error evaluator. However, in the Chien search step of the decoder, because the only points of interest are the roots of σ, there is no contribution from σ to the numerator of the error value expression. The extra factor of x is compensated for in the error calculation of the Chien search by a slight change in the exponent of α.

Along these same lines, an overall perspective of the decoding process must be kept in mind in order to see changes which can be made at various points in the algorithm. For example, it is clear from the Chien search that only the ratio of σ and ω is important; in other words, we may multiply the pair (σ,ω) by an arbitrary nonzero scalar, without affecting the calculated error pattern. Such a transformation, which we will term a *linear scaling*, can also be applied at each iteration of the key equation solution, as long as it is not inserted between the evaluate and the update phases. As we shall see, this type of modification is common to all decoding algorithms involving a Chien search and may be used to simplify the hardware implementation.

Several applications of linear scaling are possible. One aspect of the Berlekamp algorithm which may be undesirable is the multiplicative inversion of Δ required in updating the auxiliary polynomials. Suppose we choose to scale the updated σ and ω by the old value of Δ, thus cancelling the Δ^{-1} factor in τ and γ. The resulting update cycle now consists of a cross-multiply and add, with no need to do the inversion: see Figure 5-4. This version of Berlekamp's algorithm was first published by Burton [13].

Clearly, Burton's modification is but one particular instance of a more general transformation which may be applied to many decoding algorithms in an effort to minimize the hardware needed. As an additional example, which we will explore further in the following chapter, note that only one inversion is required per iteration of the key equation solver, and using bit-serial techniques the cost of such an inversion can be minimized. However, both pairs of polynomials are involved in finite-field multiplications. By scaling the error polynomials appropriately, it is possible to remove τ and γ from any non-trivial field multiplications, thus (roughly) halving the area-time complexity of the update step, in terms of multiplication count. In other words, four multiplications of a polynomial by a scalar are required to update in either the original Berlekamp algorithm or the Burton method, but note that in Figure 5-5, only two such operations are required, along with a scalar inversion and multiplication. Because the number of multiplies in the key equation solver scales as t^2, the area penalty of these scalar operations is overshadowed

```
/*          ASSUMPTIONS:                    */
/* 1. Codewords are multiples of g(x) = ∏_{k=L}^{L+2t-1} (x - α^k) . */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword. */

PROCEDURE  Berlekamp2;
BEGIN      /* no inversions during key equation solution */
                /* first calculate the syndromes */
  FOR k := 0,1,...,2t - 1 DO S_k := Σ_{i=0}^{n-1} s_i α^{i(k+L)};
  D := 0;        δ := 1;
  σ^(0)(x) := τ^(0)(x) := ω^(0)(x) := 1;
  γ^(0)(x) := 0;
  FOR k := 0,1,...,2t - 1 DO    /* solve key equation */
  BEGIN          /* evaluate (i.e., convolve) */
       Δ := Σ_{i=0}^{k} σ_i^(k) S_{k-i};
                 /* update */
       σ^(k+1) := δσ^(k) - Δxτ^(k);
       ω^(k+1) := δω^(k) - Δxγ^(k);
       IF Δ = 0 OR (2D > k + 1) THEN
            τ^(k+1) := xτ^(k);
            γ^(k+1) := xγ^(k);
       ELSE
            D := k + 1 - D;
            δ := Δ;
            τ^(k+1) := σ^(k);
            γ^(k+1) := ω^(k);
       ENDIF;
  END;
  σ(x) := σ^(2t)(x);          ω(x) := ω^(2t)(x);
  FOR k := 0,1,...,n - 1 DO     /* Chien Search */
       IF σ(α^{-k}) = 0 THEN s_k := s_k - α^{k(L-2)} ω(α^{-k})/σ'(α^{-k})  ENDIF;
END;
```

Figure 5-4. Inversionless Berlekamp Algorithm

by the area-time cost of the additional multiplies. However, this reduction is accomplished at the cost of a slightly more complex control structure, as evidenced by the additional conditional statement in Figure 5-5.

Figure 5-6 contains a synopsis of the three linear scaling transformations discussed thus far. The first example corresponds to the original Berlekamp algorithm, where there is an implicit term δ^{-1} from a previous update of $\tau(x)$. The second and third cases are related to the inversionless method of Burton and the modified algorithm just presented, respectively. Clearly,

```
/*          ASSUMPTIONS:                    */
/* 1. Codewords are multiples of g(x) = ∏_{k=L}^{L+2t-1}(x - α^k) . */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword. */

   PROCEDURE  Berlekamp3;
   BEGIN    /* no field multiplications involving γ, τ */
                    /* first calculate the syndromes */
      FOR k := 0,1,...,2t - 1 DO  S_k := Σ_{i=0}^{n-1} s_i α^{i(k+L)};
      D  :=  0;
      σ^(0)(x)  :=  τ^(0)(x)  :=  ω^(0)(x)  :=  1;
      γ^(0)(x)  :=  0;
      FOR k := 0,1,...,2t - 1 DO    /* solve key equation */
      BEGIN      /* evaluate (i.e., convolve) */
            Δ  :=  Σ_{i=0}^{k} σ_i^(k) S_{k-i};
                    /* update */
            IF Δ  =  0  THEN
                σ^(k+1)  :=  σ^(k);
                ω^(k+1)  :=  ω^(k);
            ELSE
                σ^(k+1)  :=  Δ^{-1}σ^(k) - xτ^(k);
                ω^(k+1)  :=  Δ^{-1}ω^(k) - xγ^(k);
            ENDIF;
            IF Δ  =  0  OR  (2D  >  k + 1)  THEN
                τ^(k+1)  :=  xτ^(k);
                γ^(k+1)  :=  xγ^(k);
            ELSE
                D  :=  k + 1 - D;
                τ^(k+1)  :=  Δ^{-1}σ^(k);
                γ^(k+1)  :=  Δ^{-1}ω^(k);
            ENDIF;
      END;
      σ(x)  :=  σ^(2t)(x);          ω(x)  :=  ω^(2t)(x);
      FOR k := 0,1,...,n - 1 DO     /* Chien Search */
            IF σ(α^{-k}) = 0 THEN s_k := s_k - α^{k(L-2)}ω(α^{-k})/σ'(α^{-k})  ENDIF;
   END;
```

Figure 5-5. Modified Berlekamp Algorithm

$$\begin{aligned} \sigma &:= \sigma - \Delta\delta^{-1}x\tau \\ \sigma &:= \delta\sigma - \Delta x\tau \\ \sigma &:= \Delta^{-1}\sigma - x\tau \end{aligned}$$

Figure 5-6. Examples of Linear Scaling Transformations

there are many other linear scalings which could be applied, but these three will be the most useful in practice. Although we have perhaps belabored the point in presenting three versions of the Berlekamp algorithm, the reader is encouraged to spend the time necessary to understand this class of modifications, because such transformations will be applied with impunity in several future examples.

Let us now turn our attention to decoding in the presence of erasures. As we saw in Chapter three, the ability to include soft information in a decoder can lead to large coding gains on some channels. Berlekamp presented a method of applying his key equation solver to data with known erasure locations [4, Section 10.4], but his approach requires extensive polynomial operations in addition to the key equation algorithm. It will be convenient to have an iterative technique which introduces the erasure data into the decoding algorithm in a much more natural way. Although the Berlekamp algorithm does not lend itself as readily to erasure decoding as does the Berlekamp-Massey algorithm (see the following section), in Figure 5-7 a modified version is presented that handles erasures by special initialization steps. The linear scaling transformations discussed above may be applied to this version as well.

It will be instructive to examine the differences between Figure 5-7 and the original algorithm, because similar modifications are required to incorporate erasures into the other decoding algorithms. In the presence of h erasures, the first h iterations of the key equation solver are replaced by steps which compute the error polynomials as if only erasures had occurred. That is, after the initialization iterations, $\sigma^{(h)}(x)$ is the erasure-locator polynomial, and

$$\omega^{(h)}(x) \equiv \sigma^{(h)}(x)(1 + xS(x)) \pmod{x^{h+1}}.$$

Thus, if no errors have occurred, $\omega^{(h)}$ is in fact the proper errata-evaluator polynomial. The auxiliary polynomials are similarly set up to obey the invariants of the loop, as defined in [4], Theorem 7.41; the only difference in the initialization for the auxiliary polynomials is that $\gamma^{(h)}(x)$ differs from $\omega^{(h)}(x)$ in the x^h coefficient, insuring that the pairs (σ,ω) and (τ,γ) are linearly independent. Also, the inequality (5.3) is modified to

$$\deg\left(\sigma^{(k)}(x)\right) + \deg\left(x\tau^{(k)}(x)\right) \leq k + h + 1,$$

so all expressions involving the variable D are altered to reflect this fact. Another way of explaining this change is that because the erasures have decreased the effective distance of the code to $2t + 1 - h$, only $t - h/2$ additional errors can be corrected. Note that because all updates of σ and τ are homogeneous, any polynomial which is a factor of their initialized values will also divide the final errata-locator polynomial; the erasure locations are thus guaranteed to be roots of $\sigma^{(2t)}(x)$. Once the proper initialization is completed, the remaining iterations proceed almost identically to the errors-only decoder, and the Chien search is totally unaltered. In fact, when $h = 0$ the decoder of Figure 5-7 simplifies to the original Berlekamp algorithm.

```
/*          ASSUMPTIONS:                    */
/* 1. Codewords are multiples of g(x) = ∏_{k=L}^{L+2t-1}(x - α^k) . */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword. */
/* 3. Assume h erasures occurred, at X_i for i = 0,1,...,h-1,
      and the corresponding coefficients of s(x) are set to 0. */

   PROCEDURE   BerlErase;
   BEGIN                  /* first calculate the syndromes */
      FOR k := 0,1,...,2t-1 DO S_k := Σ_{i=0}^{n-1} s_i α^{i(k+L)};
      D := h;             /* initialize for key equation solver */
      σ^(0)(x) := 1;
      FOR k := 0,1,...,h-1 DO σ^(k+1)(z) := (xX_k - 1)σ^(k)(x);
      τ^(h)(z) := σ^(h)(z);
      FOR k := 0,1,...,h DO ω_k^(h) := σ_k^(h) + Σ_{i=0}^{k-1} σ_i^(h) S_{k-1-i};
      γ^(h)(x) := ω^(h)(x) + x^h;
      FOR k := h,...,2t-1 DO    /* solve key equation */
      BEGIN              /* evaluate (i.e., convolve) */
           Δ := Σ_{i=0}^{k} σ_i^(k) S_{k-i};
                         /* update */
           σ^(k+1) := σ^(k) - Δxτ^(k);
           ω^(k+1) := ω^(k) - Δxγ^(k);
           IF Δ = 0 OR (2D > k + 1 + h) THEN
                         τ^(k+1) := xτ^(k);
                         γ^(k+1) := xγ^(k);
           ELSE
                         D := k + 1 + h - D;
                         τ^(k+1) := Δ^{-1}σ^(k);
                         γ^(k+1) := Δ^{-1}ω^(k);
           ENDIF;
      END;
      σ(x) := σ^(2t)(x);      ω(x) := ω^(2t)(x);
      FOR k := 0,1,...,n-1 DO     /* Chien search */
           IF σ(α^{-k}) = 0 THEN
                s_k := s_k - α^{-k(L-2)} ω(α^{-k}) / σ'(α^{-k});
           ENDIF;
   END;
```

Figure 5-7. Berlekamp Algorithm with Erasures

```
/*        ASSUMPTIONS:                                   */
/* 1. Codewords are multiples of g(x) = ∏_{k=L}^{L+2t-1} (x - α^k) . */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword. */
/* 3. Assume h erasures occurred, at X_i for i = 0,1,...,h-1,
      and the corresponding coefficients of s(x) are set to 0. */

PROCEDURE BerMas;
BEGIN
          /* first calculate the syndromes */
   FOR k := 0,1,...,2t-1 DO S_k := Σ_{i=0}^{n-1} s_i α^{i(k+L)};
          /* initialize for key equation solver */
   D := h;
   σ^(0)(x) := τ^(0)(x) := 1;
   FOR k := 0,1,...,h-1 DO σ^(k+1)(x) := τ^(k)(x) := X_k^{-1} σ^(k)(x) - xτ^(k)(x);
   τ^(h)(x) := σ^(h)(x);
   FOR k := h,...,2t-1 DO    /* solve key equation */
   BEGIN        /* evaluate (i.e., convolve) */
      Δ := Σ_{i=0}^{k} σ_i^(k) S_{k-i};
           /* update */
      IF Δ = 0 THEN σ^(k+1) := σ^(k)
      ELSE σ^(k+1) := Δ^{-1}σ^(k) - xτ^(k) ENDIF;
      IF Δ = 0 OR (2D > k+1+h) THEN τ^(k+1) := xτ^(k)
      ELSE
         D := k+1+h-D;
         τ^(k+1) := Δ^{-1}σ^(k);
      ENDIF;
   END;
   σ(x) := σ^(2t)(x);
   ω(x) = 0;            /* compute error evaluator */
   FOR k := 2t-1, 2t-2,...,1,0 DO ω(x) = xω(x) + Σ_{i=0}^{k} σ_i S_{k-i};
   FOR k := 0,1,...,n-1 DO    /* Chien search */
      IF σ(α^{-k}) = 0 THEN
         s_k := s_k - α^{-k(L-1)} ω(α^{-k}) / σ'(α^{-k});
      ENDIF;
END;
```

Figure 5-8. Berlekamp-Massey Algorithm with Erasures

5.4 Berlekamp-Massey Algorithm

Upon examination of the Berlekamp algorithm, it is evident that ω and γ are not involved in the evaluation criterion of the Berlekamp algorithm and their update is independent of σ and τ, implying that the error locator can be computed independently from the evaluator. Such a decoding strategy, which has come to be known as the Berlekamp-Massey algorithm, is particularly attractive for binary BCH codes where the error values are known to be 1. Even for Reed-Solomon codes, because ω can be calculated from σ and the key equation, such an approach may be advantageous, reducing both the amount of storage and the time (for a given number of finite-field arithmetic units) required to solve the key equation. The hardware tradeoffs involved in such a method will be discussed in detail in the following chapter; Cohen [19, Appendix A] has already noted that this split approach to solving the key equation is more efficient than the Berlekamp algorithm for software implementation.

Figure 5-8 presents a version of the Berlekamp-Massey algorithm that handles erasures. Observe the slight modification in the erasure initialization to make the expression for σ identical in form to the update cycle in the main key equation. Everything else up through the solution of the key equation is identical with the corresponding pseudocode from the Berlekamp algorithm, except that all statements involving ω and γ have been removed. This version of the key equation solution requires a multiplicative inversion, but note that any of the linear scaling modifications discussed above can be applied, because, once the proper errata-locator polynomial $\sigma(x)$ is determined, the error locator is calculated via the key equation (5.1); through this sequence of convolution steps, any scalar factor multiplying the error-locator polynomial is absorbed into the evaluator polynomial ω. For implementation purposes, since it is necessary to calculate inner products to solve the key equation, the hardware to compute these convolutions should already be available. Note that $\omega(x)$ in Figure 5-8 is different from the error evaluator of the previous section, since it is computed directly from the key equation instead of the modified key equation (5.4). In other words, the formal error evaluator is calculated instead of the sum of σ and $x\omega$. This difference is manifest here in the Chien search by a slight change in the exponent of a factor in the error-correction term, owing to the removal of the x term which multiplies the evaluator polynomial in the original Berlekamp algorithm.

5.5 Euclidean Decoding Algorithm

Euclid's algorithm is a recursive procedure for finding the greatest common divisor (gcd) of two elements of a unique factorization domain, such as the integers or the ring of polynomials over a field. For decoding applications we are interested in the polynomials over finite fields, and our development of the algorithm in this section follows closely that given in [40, Section 8.4]. In general, if $d(x) = \gcd(a(x), b(x))$, there exist polynomials $u(x)$ and $v(x)$ such that

$$a(x)u(x) + b(x)v(x) = d(x).$$

It turns out that Euclid's algorithm can be extended to produce not only $d(x)$, but also the auxiliary polynomials $u(x)$ and $v(x)$. At each iteration of the gcd algorithm a new estimate of each of these polynomials is obtained, based on the previous values.

Initially, the estimates are given by

$$\begin{aligned} u_0(x) &= 1, \quad & v_0(x) &= 0, \quad & r_0(x) &= a(x), \\ u_1(x) &= 0, \quad & v_1(x) &= 1, \quad & r_1(x) &= b(x). \end{aligned}$$

Let us denote by $q_i(x)$ the quotient of $r_{i-2}(x)$ and $r_{i-1}(x)$; in other words, choose $q_i(x)$ such that

$$\deg(r_{i-2}(x) - q_i(x)r_{i-1}(x)) < \deg(r_{i-1}(x)). \tag{5.5}$$

Then the new estimates for the polynomials are given by

$$\begin{aligned} r_i(x) &= r_{i-2}(x) - q_i(x)\, r_{i-1}(x) \\ u_i(x) &= u_{i-2}(x) - q_i(x)\, u_{i-1}(x) \\ v_i(x) &= v_{i-2}(x) - q_i(x)\, v_{i-1}(x). \end{aligned} \tag{5.6}$$

Note that $r_i(x)$ is the remainder when $r_{i-2}(x)$ is divided by $r_{i-1}(x)$. According to (5.5), the degrees of the remainders $r_i(x)$ are strictly decreasing, and it turns out that the last nonzero remainder, say $r_n(x)$, is the greatest common divisor of $a(x)$ and $b(x)$. Further, using (5.6) it can be shown by induction that for $0 \leq i \leq n$,

$$u_i(x)a(x) + v_i(x)b(x) = r_i(x),$$

which implies that $u(x) = u_n(x)$ and $v(x) = v_n(x)$.

Although polynomial division is a rather complex operation, it is actually composed of a series of simpler steps, each consisting of multiplying $r_{i-1}(x)$ by a scalar multiple of a monomial and adding the result to $r_{i-2}(x)$ to cancel the latter's leading coefficient. This basic iteration is repeated until the degree of r_{i-2} falls below that of r_{i-1}, at which point the remainder is in fact r_i. Then, the role of the two polynomials is reversed and the procedure continues. If exactly the same operations are performed on the u_i and the v_i polynomials, the appropriate values of the auxiliary polynomials are generated. Thus, it is clear that Euclid's algorithm can be implemented with fairly simple hardware.

In the Berlekamp algorithm, the key equation (5.1) is viewed as an exercise in shift-register synthesis, but it can also be interpreted as a problem in rational approximation. If we consider the syndromes to be the initial coefficients of a power series

$$P(x) = S_0 + S_1 x + S_2 x^2 + \cdots + S_{2t-1}x^{2t-1} + \cdots,$$

then the goal is to find two polynomials whose quotient is identical to $P(x)$ in the first $2t$ terms, subject to the constraint that the sum of the degrees of the two polynomials is less than $2t$. Rational functions of this type are known as Padé approximants, and it can be shown that all such pairs of polynomials are generated at some stage of Euclid's extended algorithm [40, Theorem 8.5]. From above, if we let $a(x) = x^{2t}$ and $b(x) = S(x)$ and apply the gcd procedure, at some step we will find

$$u_i(x)x^{2t} + v_i(x)S(x) = r_i(x),$$

where $\deg(r_i) < t$. Another way of looking at this equation is

$$v_i(x)S(x) \equiv r_i(x) \pmod{x^{2t}}.$$

Clearly we would like to make the correspondence $\sigma(x) = v_i(x)$ and $\omega(x) = r_i(x)$. Since all Padé approximants are produced during Euclid's algorithm, if a proper solution to the key equation

```
/*          ASSUMPTIONS:                    */
/* 1. Codewords are multiples of g(x) = ∏_{k=L}^{L+2t-1}(x - α^k) . */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword. */
/* 3. Assume h erasures occurred, at X_i for i = 0,1,...,h - 1,
      and the corresponding coefficients of s(x) are set to 0. */
/* 4. (a,b) := (b,a) means to swap the contents of the polynomials. */
/* 5.  lead(a) returns the leading coefficient of a(x). */

PROCEDURE Euclid;
BEGIN
            /* first calculate the syndromes */
   FOR k := 0,1,...,2t - 1 DO S_k := Σ_{i=0}^{n-1} s_i α^{i(k+L)};
   r(x) := x^{2t};          /* initialize for gcd computation */
   ω(x) := Σ_{i=0}^{2t-1} S_i x^i;
   v(x) := 0;     σ(x) = 1;
   FOR k := 0,1,...,h - 1 DO
   BEGIN    /* initialize for erasures */
          ω(x) := (xX_k - 1)ω(x) (mod x^{2t});
          σ(x) := (xX_k - 1)σ(x);
   END;
            /* perform Euclid's algorithm on ω and r */
   WHILE deg(ω) ≥ t + h/2 DO       /* evaluate */
   BEGIN
          d := deg(r) - deg(ω);
          p := lead(r)/ lead(ω);
          r(x) := r(x) - px^d ω(x);       /* update */
          v(x) := v(x) - px^d σ(x);
          IF deg(r) < deg(ω) THEN
               (r,ω) := (ω,r);          /* swap */
               (v,σ) := (σ,v);
          ENDIF;
   END;
   FOR k := 0,1,...,n - 1 DO     /* Chien search */
       IF σ(α^{-k}) = 0 THEN
            s_k := s_k - α^{-k(L-1)} ω(α^{-k})/σ'(α^{-k});
       ENDIF;
END;
```

Figure 5-9. Euclid's Algorithm with Erasures

exists, it will be found by applying the gcd recursion. In fact, it can be shown that the proper final iteration is exactly that where the degree of r_i first drops below t; the error locator is then assigned to be $v_i(x)$, and $r_i(x)$ is the error evaluator.

Figure 5-9 presents a decoding algorithm which employs the gcd computation to solve the key equation. The case without erasures (i.e., $h = 0$) proceeds exactly as described above. However, with the addition of erasure data, decoding becomes somewhat more complex. Note that the key equation (5.1) still holds in the presence of erasures, but for implementation it must be interpreted somewhat differently. Namely, let $\lambda(x)$ be the erasure-locator polynomial, and $\sigma(x)$ be the error locator. The key equation then becomes

$$\lambda(x)\sigma(x)S(x) \equiv \omega(x) \pmod{x^{2t}}, \tag{5.7}$$

where both $\lambda(x)$ and $S(x)$ are known. Hence we can define a modified syndrome polynomial $T(x) \equiv \lambda(x)U(x) \pmod{x^{2t}}$, so that

$$\sigma(x)T(x) \equiv \omega(x) \pmod{x^{2t}}. \tag{5.8}$$

Then (5.8) is solved for σ and ω, and a Chien search can be performed using $\Lambda(x) = \lambda(x)\sigma(x)$ as the errata-locator polynomial and $\omega(x)$ as the errata evaluator. However, note from (5.7) that the upper bound on $\deg(\omega)$ is no longer t. Instead, since at most $t - h/2$ errors can be corrected in addition to the erasures, we have $\deg(\sigma) \leq t - h/2$, implying

$$\deg(\omega) < \deg(\Lambda) = \deg(\lambda) + \deg(\sigma) \leq h + t - h/2 = t + h/2,$$

which is exactly the terminating condition on the key equation loop of Figure 5-9. Each iteration decreases the degree of ω by at least one, so, because of the swapping of polynomials, up to

$$1 + \deg(r_0(x)) - (t + h/2) + \deg(r_1(x)) - (t + h/2) \leq 1 + 4t - 1 - 2t - h = 2t - h$$

iterations may be required to complete the algorithm. Thus, counting the h erasure initialization steps, there are a maximum of $2t$ iterations involved in solving the key equation, just as we have seen in previous algorithms. Observe, however, that the algorithm may terminate in fewer than $2t$ steps, unlike any other decoder we will encounter.

In Figure 5-9, erasures are handled roughly as described in the previous paragraph, with the implementation detailed at a lower level. The transformation of $S(x)$ into $T(x)$ is performed one erasure at a time, multiplying by $(xX_i - 1)$ and retaining only terms of degree less than $2t$. This operation involves multiplying the polynomial coefficients by a scalar and adding adjacent terms, features which should already be available for use in the key equation loop; thus, again, the erasure initialization fits well into the hardware decoding scheme. Calculation of the errata-locator polynomial $\Lambda(x)$ in Figure 5-9 takes advantage of the homogeneity of the update stage for the locator polynomial: because $v_0(x) = 0$, any polynomial which divides $v_1(x)$ will also be a divisor of all further iterates of both $\sigma(x)$ and $v(x)$. So, instead of computing $\sigma(x)$ separately and then multiplying by the erasure locator, the auxiliary polynomial $v_1(x)$ is set equal to the erasure locator $\lambda(x)$, using (almost) the same initialization operations as are applied to $S(x)$. However, because $\deg(v(x)) = h \leq 2t$ after the erasure setup phase, a $\pmod{x^{2t}}$ operation on $v(x)$ has no effect unless $h = 2t$, and this latter case entails erasure-only decoding, where it is imperative that $\deg(v(x)) = 2t$ so that all the erasures can be located in the Chien search.

This method of erasure initialization is attractive because it involves simple operations which are applied consistently to both the remainder and the auxiliary polynomials, just as occurs in the remaining portions of the key equation solution.

Notice that, as in the Berlekamp algorithm, the error-locator and error-evaluator polynomials are updated totally independently. However, since the evaluation criterion depends only on the evaluator polynomials, it is only possible here to calculate $\omega(x)$ without computing $\sigma(x)$. While the error locator could then be extracted using the key equation, such an operation requires a convolution step, which is not otherwise necessary for the gcd computation. Thus, splitting the Euclidean key equation solution into two phases does not appear to be as attractive as in the Berlekamp-Massey algorithm.

In the Euclidean algorithm, the evaluation criterion involves only the degree of the remainder polynomial, and the update step requires only the degree and the leading coefficients of the two remainder polynomials. However, while it is simple conceptually to find the degree of a polynomial, extracting this information in hardware is not as straightforward. Nonetheless, in terms of communication between coefficients, this algorithm has perhaps the least complex evaluate-update cycle of any decoder, and this simplicity can be exploited to design a very modular decoder. In particular, if we consider the polynomials as a stream of coefficients (high order first), it is clear that each iteration of the algorithm, including the erasure initialization, can be implemented by a single stage which transforms the incoming stream into the appropriate outgoing stream. Each unit can latch onto the leading coefficient of the incoming polynomials and determine their degree from the delay until the first nonzero coefficient arrives. A pipelined linear array of such stages could be used to solve the key equation, and in Chapter six we will discuss the architecture of such a decoder, which was first proposed by Kung [14].

5.6 Berlekamp-Welch Algorithm

All of the decoding algorithms discussed thus far begin by calculating the power-sum syndromes, and we have seen that there are many variations of each key equation solution technique. Now we turn our attention to decoding methods which do not explicitly compute the syndromes. For example, since α^{L+k} is a root of $g(x)$ for $k = 0, 1, \ldots, 2t-1$, the syndrome $S_k = s(\alpha^{L+k}) = r(\alpha^{L+k})$, where the remainder polynomial $r(x) \equiv s(x) \pmod{g(x)}$, so it is clear that all of the syndromes can be generated from $r(x)$. In other words, the remainder polynomial contains all the information necessary to decode the received word, and one might hope to decode working directly with the remainder, without the need to compute the syndromes. Such an approach would be particularly attractive in a system which is used both for receiving and transmitting data (i.e., reading and writing), where there is a need both to encode and to decode. The encoder could then do double duty, generating the appropriate parity characters for transmission and eliminating the need for a syndrome generator by calculating the remainder polynomial for use in decoding the received word.

Berlekamp and Welch proposed such an algorithm in 1982, and a crucial step in coming up with the procedure was finding a key equation (5.2) which involves only the remainder coefficients. A brief description of their derivation of this version of the key equation will be given here; for more details refer to [8]. First, suppose that only one error has occurred, at a message location given by X, with error value Y. Then, for $k = L, L+1, \ldots, L+2t-1$,

$$S_{k-L} = YX^k = s(\alpha^k) = r(\alpha^k).$$

So, for $k = L+1, L+2, \ldots, L+2t-1$,

$$r(\alpha^k) - Xr(\alpha^{k-1}) = YX^k - XYX^{k-1} = 0,$$

implying that the polynomial $u(x) = r(x) - Xr(\alpha^{-1}x)$ has roots at the locations

$$\alpha^{L+1}, \alpha^{L+2}, \alpha^{L+3}, \ldots, \alpha^{L+2t-2}, \alpha^{L+2t-1}.$$

Thus, $u(x)$ is divisible by the polynomial

$$p(x) = \prod_{k=L+1}^{L+2t-1} (x - \alpha^k) = \sum_{i=0}^{2t-1} p_i x^i,$$

which is in fact the generator polynomial for a Reed-Solomon code of distance $2t$. Now, the degree of $p(x)$ is $2t-1$, and the degree of $u(x)$ is less than $2t$, so clearly $u(x)$ is a scalar multiple of $p(x)$. Equating coefficients, we find

$$u_i = r_i(1 - X\alpha^{-i}) = Ap_i, \tag{5.9}$$

for some constant A. Notice that if the error occurred at check (i.e., parity) location $X = \alpha^j$ for $0 \leq j < 2t$, then $r(x) = Yx^j$, and $A = 0$. Evidently the check positions will be treated differently from message locations in this approach; in fact, since $A = 0$ we will find that (5.9) gives no information about check error values, so errors in the parity locations must be corrected by reencoding.

For a single error, let us define the error-locator polynomial as $W(x) = x - X$. Then, by (5.9), it is clear that for $i = 0, 1, \ldots, 2t-1$, we have

$$r_i W(\alpha^i) = A\alpha^k p_k. \tag{5.10}$$

As long as X is not a check location, $W(\alpha^i)$ is nonzero for these values of i, so (5.10) can be solved for r_i in terms of $W(x)$ and A. To find the error values, we note that

$$Y = X^{-L} r(\alpha^L) = X^{-L} \sum_{i=0}^{2t-1} r_i \alpha^{iL} = AX^{-L} \sum_{i=0}^{2t-1} \frac{p_k \alpha^{i(L+1)}}{\alpha^i - X} = Af(X), \tag{5.11}$$

where $f(X)$ is the indicated function of X which can be precomputed. In terms of this function, the remainder coefficients are then given by

$$r_i = \frac{Y p_i \alpha^i}{f(X)(\alpha^i - X)}.$$

When multiple message errors have occurred, say of value Y_j at location X_j for $j = 1, 2, \ldots, e$, then by linearity

$$r_i = p_i \alpha^i \sum_{j=1}^{e} \frac{Y_j}{f(X_j)(\alpha^i - X_j)}, \tag{5.12}$$

```
/*          ASSUMPTIONS:                                    */
/* 1. Codewords are multiples of g(x) = ∏_{k=L}^{L+2t-1} (x - α^k) . */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword. */
/* 3. The remainder is r(x) = Σ_{k=0}^{2t-1} r_k x^k ≡ s(x) (mod g(x)). */
/* 4. The constants p_k are defined by ∏_{i=L+1}^{L+2t-1} (x - α^i) = Σ_{i=0}^{2t-1} p_i x^i. */
/* 5. The constants f(X) = α^{t(2t-1)} α^{2tL} X^{-L} g'(α^L)/g(X α^L). */
/* 6. Assume h message erasures occurred, at X_i for i = 0,1,...,h-1,
      and the corresponding coefficients of s(x) are set to 0. */
/* 7. Assume check erasures occurred where erased(s_k) is TRUE. */

    PROCEDURE  WelchBerl;
    BEGIN
      N^(0)(x) := V^(0)(x) := 0;     M^(0)(x) := W^(0)(x) := 1;
      FOR k := 0,1,...,h-1 DO W^(0)(x) := (x - X_k) W^(0)(x);
      FOR k := 0,1,...,2t-1 DO          /* solve key equation */
          IF  NOT erased(s_k)
           THEN                          /* evaluate */
              a_k := α^k p_k N^(k)(α^k) - r_k W^(k)(α^k);
              b_k := α^k p_k M^(k)(α^k) - r_k V^(k)(α^k);
              IF a_k = 0 THEN b_k := 1 ENDIF;
              IF deg(a_k M^(k)) < deg(b_k W^(k)) THEN   /* update */
                  N^(k+1)(x) := b_k N^(k)(x) - a_k M^(k)(x);
                  W^(k+1)(x) := b_k W^(k)(x) - a_k V^(k)(x);
                  M^(k+1)(x) := (x - α^k) M^(k)(x);
                  V^(k+1)(x) := (x - α^k) V^(k)(x);
              ELSE
                  M^(k+1)(x) := b_k N^(k)(x) - a_k M^(k)(x);
                  V^(k+1)(x) := b_k W^(k)(x) - a_k V^(k)(x);
                  N^(k+1)(x) := (x - α^k) N^(k)(x);
                  W^(k+1)(x) := (x - α^k) W^(k)(x);
              ENDIF;
          ENDIF;
      W(x) := W^(2t)(x);      N(x) := N^(2t)(x);
      FOR k := 2t, 2t+1, ..., n-1 DO     /* Chien search */
          IF W(α^k) = 0 THEN   /* only correct message characters */
             s_k := s_k - f(α^k) N(α^k)/W'(α^k) ;
          ENDIF;
    END;
```

Figure 5-10. Berlekamp-Welch Algorithm with Erasures

if and only if s_i is a correct check digit. If we consider the rational function

$$F(x) = \sum_{j=1}^{e} \frac{Y_j}{f(X_j)(x - X_j)}, \tag{5.13}$$

it is clear that there exists a polynomial $N(x)$ of degree less than e such that $F(x) = N(x)/W(x)$. The error locations are determined by the poles of $F(x)$, and the error values are

$$Y_j = f(X_j)\frac{N(X_j)}{W'(X_j)}.$$

Thus, these two polynomials $N(x)$ and $W(x)$ are quite analagous to the error-evaluator and error-locator polynomials which we derived in the syndrome case. From (5.12) and (5.13), the object is to find two polynomials such that

$$p_i \alpha^i N(\alpha^i) = r_i W(\alpha^i) \quad \text{for} \quad i = 0, 1, \ldots, 2t-1, \tag{5.14}$$

which is equivalent in form to the version of the key equation (5.2) given earlier in the chapter. Since at most t errors can be decoded, we must have $\deg(N) < \deg(W) = e \leq t$.

Given the complexity of the above derivation, it is perhaps not surprising that remainder decoding methods took so long to be discovered: merely finding the appropriate problem to solve is involved enough. Figure 5-10 presents the original version of the Berlekamp-Welch algorithm, complete with erasure handling. We will attempt to point out the many similarities to the syndrome decoding techniques, along with a few differences. The remainder coefficients are not explicitly computed in this presentation, so the three major loops are: erasure initialization, key equation solution, and Chien search. Observe that there are again four polynomials, the locator W and the evaluator N, along with a corresponding auxiliary pair V and M. In the first loop $W(x)$ is initialized to be the message erasure-locator polynomial; by the homogeneity of the update step, and the fact that the associated auxiliary polynomial $V^{(0)}(x)$ is zero, it is clear that the erasure locations will be roots of the final errata locator. In the key equation solver loop, each coefficient of the remainder polynomial is tested to see if the current estimate of the error polynomials satisfies (5.14). It should be noted that the presence of message erasures does not decrease the number of iterations of the key equation solver, unlike in the other decoding algorithms; however, only those coefficients corresponding to non-erased check locations are processed.

The evaluation criterion in the Berlekamp-Welch algorithm consists of a polynomial evaluation at the next check location to calculate the discrepancy in (5.14) using the current estimates of the error polynomials. In Figure 5-11, the check locations are tested sequentially beginning with α^0, but actually the ordering can be arbitrary. If the discrepancy is zero, the current estimates are retained and the auxiliary polynomials are slightly modified. Otherwise, the current estimates are updated in a manner guaranteed to cancel the discrepancy and minimize the degree of $W(x)$. Thus, this key equation solver is very similar in flavor to the other algorithms we have studied. However, because the evaluation criterion involves all four polynomials, there is no obvious way to split the algorithm, as was possible in the Berlekamp-Massey and Euclid approaches. Also, in this version of the algorithm, it is necessary to test the degrees of the polynomials W and M, which may not be particularly easy to implement in hardware, as we mentioned previously. Fortunately, one can prove that, after j non-erased check locations have

```
/*        ASSUMPTIONS:                                    */
/* 1. Just as in the previous example, with the following exception: */
/* 2. The constants f(X) = α^{t(2t-1)} α^{2tL} X^{1-L} g'(α^L)/g(Xα^L). */

    PROCEDURE BerlWelchMod;
    BEGIN
      N^(0)(x) := V^(0)(x) := 0;
      M^(0)(x) := W^(0)(x) := 1;
      q := D := h;
      FOR k := 0,1,...,h-1 DO W^(0)(x) := (x - X_k)W^(0)(x);
      FOR k := 0,1,...,2t-1 DO          /* solve key equation */
          IF erased(s_k) THEN
              q := q - 1;
              N^(k+1)(x) := N^(k)(αx); W^(k+1)(x) := W^(k)(αx);
              M^(k+1)(x) := M^(k)(αx); V^(k+1)(x) := V^(k)(αx);
          ELSE                          /* evaluate */
              a_k := α^k p_k N^(k)(1) - r_k W^(k)(1);
              b_k := α^k p_k M^(k)(1) - r_k V^(k)(1);
              IF a_k = 0 THEN b_k := 1;
              IF a_k = 0 OR (b_k ≠ 0 AND 2D > k + q) THEN   /* update */
                  N^(k+1)(x) := b_k N^(k)(αx) - a_k M^(k)(αx);
                  W^(k+1)(x) := b_k W^(k)(αx) - a_k V^(k)(αx);
                  M^(k+1)(x) := (αx - 1)M^(k)(αx);
                  V^(k+1)(x) := (αx - 1)V^(k)(αx);
              ELSE
                  M^(k+1)(x) := b_k N^(k)(αx) - a_k M^(k)(αx);
                  V^(k+1)(x) := b_k W^(k)(αx) - a_k V^(k)(αx);
                  N^(k+1)(x) := (αx - 1)N^(k)(αx);
                  W^(k+1)(x) := (αx - 1)W^(k)(αx);
                  D := D + 1;
              ENDIF;
          ENDIF;
      W(x) := W^(2t)(x);      N(x) := N^(2t)(x);
      FOR k := 2t, 2t+1, ..., n-1 DO BEGIN    /* Chien search */
          IF W(1) = 0 THEN s_k := s_k - f(α^k) N(1)/W'(1) ENDIF;
          W(x) := W(αx);      N(x) := N(αx);
      END;
    END;
```

Figure 5-11. Modified Berlekamp-Welch Algorithm with Erasures

been processed, $\deg(M) + \deg(W) = j + h$. Thus, as we shall see below, it is possible to implement this decision with a variable which tracks the degree of W, similar to the parameter D in the original Berlekamp algorithm. Once the key equation solution is complete, the Chien search proceeds just as in previous examples, except that the constant $f(\alpha^k)$ multiplying the error value is no longer just a power of α. Although Liu has shown that the expression for the function f can be simplified somewhat (as in Figure 5-11) from the form given above in equation (5.11), it is nonetheless expensive to compute on the fly and would perhaps best be stored in a ROM.

In remainder decoding methods, the application of linear scaling can be coupled with other modifications to produce a radical change in the algorithm's appearance. Notice that in the Berlekamp-Welch decoder the polynomials are evaluated at all elements of the field in turn; this sequence of evaluations is difficult to implement efficiently in parallel. For example, to evaluate $W(x)$ at α^k, the coefficient of x^i must have an associated register which contains the element α^{ik}. This register must be capable of being multiplied by the coefficient W_i and then multiplying itself by α^i to prepare for the next point. Such a scheme requires either an additional multiplier or the sharing of an existing multiplier. Along the same lines, observe that in the update step there are possibly two multiplications involving each polynomial coefficient: one by either a_k or b_k, and the other by α^k.

Both of these difficulties can be circumvented by the use of linear scaling and by applying an operator $\mathcal{D} : a(x) \mapsto a(\alpha x)$ at each step, as is shown in Figure 5-11. In other words, in terms of the original version of the algorithm, the new polynomials are changed to $\tilde{a}^{(k)}(x) = a^{(k)}(\alpha^k x)$, so that all polynomials are evaluated only at $x = 1$ throughout the algorithm. For this reason it is necessary to apply $\mathcal{D}$ even when processing erased check positions. The operator $\mathcal{D}$ can be implemented very efficiently, since each coefficient needs only to be multiplied by a (hard-wired) constant; this change can occur during the evaluation phase, so that during the update step each coefficient needs only to be multiplied by either a_k or b_k. Under the operator $\mathcal{D}$, multiplication by $(x - \alpha^k)$ during the update of iteration k becomes (with the help of a little linear scaling) multiplication by $(\alpha x - 1)$, which can be implemented without a multiplier, since the bits of αx are readily available if x is expressed in a canonical dual basis. Note that the Chien search uses $\mathcal{D}$ to step through the message locations in sequence, always evaluating at $x = 1$; such a transformation introduces a spurious factor into the derivative, which accounts for the slight difference between the definition of $f(X)$ here and in the original algorithm. This technique is in fact an extension of the root searching method originally proposed by Chien for hardware implementation of BCH codes [17], and we will discuss its ramifications in Chapter six. Also, as mentioned previously, D is the degree of $W(x)$, and the condition $2D > k + q$ holds if and only if $\deg(W) > \deg(M)$, since q keeps track of the difference between the number of message erasures and the number of processed check erasures. Given these facts, it is clear that the modified algorithm of Figure 5-11 is functionally identical to the original Berlekamp-Welch method.

5.7 Liu Algorithm

Attempting to overcome some of the motivational and computational shortcomings (as mentioned above) of the original Berlekamp-Welch algorithm, T. H. Liu, one of Welch's graduate students, derived another remainder decoding technique which is presented in Figure 5-12 for the errors-only case [35]. From the evaluation phase of the Liu algorithm, it is clear that the key equation of interest is

$$q_i N(\alpha^i) = r_i W(\alpha^i), \tag{5.15}$$

```
/*          ASSUMPTIONS:                                          */
/* 1. Codewords are multiples of g(x) = ∏_{k=L}^{L+2t-1}(x - α^k). */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword. */
/* 3. The remainder is r(x) = Σ_{k=0}^{2t-1} r_k x^k ≡ s(x) (mod g(x)). */
/* 4. The constants q_k are defined by ∏_{i=L}^{L+2t-2}(x - α^i) = Σ_{i=0}^{2t-1} q_i x^i. */
/* 5. The constants f(X) = α^{(t-1)(2t-1)} α^{2tL} X^{-L} g'(α^L)/g(Xα^L). */
/* 6. (a,b) := (b,a) means to swap the contents of the polynomials. */

PROCEDURE Liu;
BEGIN
   M^(0)(x) := W^(0)(x) := 1;
   V^(0)(x) := x;   N^(0)(x) := 0;
   D := 0;
   FOR k = 0,1,...,2t - 1 DO     /* solve key equation */
   BEGIN          /* evaluate */
      a_k := q_k N^(k)(α^k) - r_k W^(k)(α^k);
      b_k := q_k M^(k)(α^k) - r_k V^(k)(α^k);
      IF a_k = 0 THEN     /* update */
         M^(k+1)(x) := (x - α^k) M^(k)(x);
         V^(k+1)(x) := (x - α^k) V^(k)(x);
      ELSE
         N^(k+1)(x) := b_k N^(k)(x) - a_k M^(k)(x);
         W^(k+1)(x) := b_k W^(k)(x) - a_k V^(k)(x);
         M^(k+1)(x) := (x - α^k) N^(k)(x);
         V^(k+1)(x) := (x - α^k) W^(k)(x);
         IF 2D < k + 1 THEN   /* swap */
            (N^(k+1), M^(k+1)) := (M^(k+1), N^(k+1));
            (W^(k+1), V^(k+1)) := (V^(k+1), W^(k+1));
            D := D + 1;
         ENDIF;
      ENDIF;
   END;
   W(x) := W^(2t)(x);          N(x) := N^(2t)(x);
   FOR k := 2t, 2t + 1,...,n - 1 DO     /* Chien search */
      IF W(α^k) = 0 THEN   /* only correct message characters */
         s_k := s_k - f(α^k) N(α^k)/W'(α^k);
      ENDIF;
END;
```

Figure 5-12. Liu Decoding Algorithm

where the q_i are coefficients of a generator polynomial very similar to $g(x)$ in the Berlekamp-Welch algorithm. However, observe from the definitions of these two polynomials that the roots of $q(x)$, as viewed in the frequency domain, are shifted one location. Thus, from the frequency translation properties of the Fourier transform, and the fact that the coefficients of $q(x)$ form a Reed-Solomon codeword in a code of minimum distance $2t$, it can easily be shown that

$$q_i = \alpha^{i-(2t-1)} p_i,$$

so Liu's key equation (5.15) is identical to (5.14), up to the constant $\alpha^{-(2t-1)}$. In other words, the error-evaluator polynomial in Figure 5-12 differs from that of the Berlekamp-Welch algorithm by a factor of α^{2t-1}, but note that this discrepancy is compensated for by the definition of $f(X)$. The parameter D in the Liu algorithm tracks the degree of $W(x)$. Clearly a set of modifications similar to those of Figure 5-11, involving the operator $\mathcal{D}$, could be applied to this algorithm as well. Again, because the evaluation criterion depends on both sets of polynomials, there is no obvious way to split the algorithm, computing the locator without the evaluator. Evidently, the flow of the Liu algorithm is virtually identical to the Berlekamp-Welch method, except for small changes in initialization and in one of the update branches, so any differences in computational complexity between the two remainder methods are only second-order effects. One is led to conjecture that there exists a simple transformation which would show the equivalence of the two algorithms, but thus far such a proof has been elusive.

Unlike the last few decoders presented, Figure 5-12 does not include erasure decoding, for two reasons. First, it is not yet well understood what are the proper initialization steps for the Liu algorithm to operate correctly in the presence of erasures. Although it probably would not be overly difficult to extend this method to handle erasures, either by analogy with the Berlekamp-Welch approach or from a study of the invariants of the key equation solver loop, Liu has opted for an altogether different approach, known as dynamic check allocation, which was originally proposed (and implemented) by Berlekamp and Welch in the paper detailing their algorithm [8]. A brief description of their method is described in the following paragraphs.

One interesting facet of remainder decoding is that, given a correctable error pattern, if all the errors occur in check positions, the weight of the remainder polynomial is less than t, and the error pattern is the remainder polynomial itself. Such a condition can be detected and then corrected by inspection; there is no corresponding result when the power-sum syndromes are used. In the Berlekamp-Welch algorithm, solution of the key equation (5.14) provides no information about the error values in the check locations; this limitation also holds in the Liu algorithm, so any check errors must in general be corrected by reencoding.

Because Reed-Solomon codes are maximum distance separable, any set of $2t$ positions can serve as parity (or check) locations. Given remainder coefficients for the standard set of check positions, a simple linear transformation, involving LaGrangian interpolation polynomials, can be used to map to corresponding coefficients for the new parity set; essentially this mapping can be thought of as a basis change. A derivation of the linear transformation is beyond the scope of this thesis, and the interested reader should refer to [8] or [35] for details. However, it should be noted that calculation of the matrix entries is non-trivial, since there are $4t^2$ such elements, each requiring the evaluation of a polynomial of degree $2t-1$; similarly, application of the linear transformation itself requires a large amount of computation.

Now, if all the errors occurred in the new check locations, they could be corrected as described above. So, one approach to erasure decoding is to change the parity locations to include

all erasures, apply the decoding algorithm, which involves only non-erased check positions, and correct the erasures by reencoding the new parity locations. In other words, the check positions can be allocated as needed. There are many coding channels for which such an approach may be very efficient, despite the large overhead associated with the matrix transformation. For example, in a multi-track magnetic tape system, with each track corresponding to one character position in the codewords, typical errors consist of very long error bursts on one or more tracks. If the decoder knows that a particular track is currently in the middle of an error burst, the track can be allocated as a check track, so that all subsequent errors appear as check errors and can be corrected by inspection. For such an approach to be helpful, the burst length must typically be much longer than the time required to compute the transform matrix.

Clearly this philosophy assumes that the decoder cannot keep up with worst-case error rates. While much work has been done on average-case optimized decoders [19], our goal in this thesis is to study decoding systems which output corrected data in synchrony with the incoming data, removing the need for elastic buffers and other clever enhancements which invariably increase system complexity. Given this perspective (which may not be valid in all applications), the inability of the Liu algorithm to handle erasures without massive matrix computations must be regarded as a serious detriment. For example, in the amount of area required to implement a matrix multiplier so that erasures can be corrected with little overhead, a high-performance decoder could probably be built which would correct the erasures directly.

Because the Liu algorithm is so similar to the Berlekamp-Welch technique, it would appear that the major contribution of Liu is more in his derivation method than in the resulting algorithm. For example, the definition of the function $f(X)$ originally given by Berlekamp and Welch is computationally rather cumbersome, as evidenced by (5.11). However, by investigating the question of error evaluation from a different point of view, Liu was able to come up with a much more tractable form, which we have included in the presentation of both algorithms. Also, while the derivation of the key equation (5.14) due to Berlekamp and Welch is clear, it gives little indication of the relationship between remainder decoding and the more familiar approach involving the power-sum syndromes. By contrast, Liu arrived at his key equation starting from the Massey viewpoint of the syndrome key equation as a linear recurrence relation. Suppose that e errors have occurred, and that $\sigma(x)$ is the appropriate error-locator polynomial. If $\sigma(x)$ is normalized so that $\sigma_0 = 1$, then the recurrence relation is given by

$$S_k = \sum_{i=1}^{e} \sigma_i S_{k-i} \quad \text{for} \quad k = e, e+1, \ldots, 2t-1.$$

Equivalently, there is the more general form

$$\sum_{i=0}^{e} \sigma_i S_{k-i} = 0, \tag{5.16}$$

which imposes no restriction on the normalization of σ. Let us assign $W(x)$ to be the polynomial reciprocal to the error locator; in other words, $W_i = \sigma_{e-i}$, so the roots of W are the inverses of the roots of σ. In terms of W, after a suitable change of indices, (5.16) becomes

$$\sum_{i=0}^{e} W_i S_{j+i} = \sum_{i=0}^{e} W_i r(\alpha^{L+i+j}) = 0 \quad \text{for} \quad j = 0, 1, \ldots, 2t-1-e. \tag{5.17}$$

At this point, Liu introduced the operator D which was employed above in a modified version of the Berlekamp-Welch algorithm. Given a polynomial $a(x)$, $D^i a(x) = a(\alpha^i x)$, so it makes sense to talk about a polynomial in D. In particular, consider the polynomial operator $W(D) = \sum_{i=0}^{e} W_i D^i$ and observe that according to (5.17), $W(D)r(x)$ has zeroes at α^{L+j} for $j = 0, 1, \ldots, 2t-1-e$. That is, $W(D)r(x)$ is a multiple of the polynomial $\prod_{k=L}^{L+2t-1-e}(x-\alpha^k)$. From here, Liu goes on to show that there exists a polynomial $N(D)$ such that $W(D)\,r(x) = N(D)\,q(x)$, where $q(x)$ is as defined in Figure 5-12; by equating coefficients, this result can be simplified to yield the key equation. Again, the specific details of Liu's derivation are not relevant here.

The main point of interest, aside from the important result of relating remainder methods to syndrome decoding, is that this approach gives some hope of splitting the algorithm to produce only the error-locator polynomial. It seems feasible to construct a procedure, by analogy with the Berlekamp-Massey algorithm, which iteratively generates $W(D)$ directly from the remainder polynomial, without resorting to computing $N(D)$ also. Although such a method might have some practical advantages over the full Liu algorithm, after some thought it becomes clear that each iteration of this algorithm would involve evaluating a polynomial, which is exactly the computation we are trying to avoid by not using the power-sum syndromes. As we mentioned earlier, the area cost of an encoder is not tremendously less than that of a syndrome generator. In other words, a split remainder decoder would probably begin to look suspiciously like the Berlekamp-Massey decoder itself. As we might expect, each decoding algorithm gives us greater insight into the generalizations which might be employed to produce still more decoding algorithms.

5.8 Blahut's Time-Domain Decoder

The power-sum syndromes actually give a window into the frequency spectrum of the error pattern e_i; in the notation of this chapter, $S_k = E_{L+k}$. Applying the Berlekamp-Massey key equation solver to these syndromes yields the error-locator polynomial. Given $\sigma(x)$, the entire error spectrum can then be reconstructed using the recurrence relation (5.16):

$$E_k = \sum_{i=1}^{e} \sigma_i E_{k-i}. \tag{5.18}$$

Once the spectrum $\mathbf{E}$ is known, an inverse transform produces the error pattern $\mathbf{e}$, allowing the received word to be corrected. Although this approach comprises a valid decoding algorithm, observe that it entails part of a forward transform (syndrome computation) and an inverse transform, with the key equation being solved in the frequency domain. Blahut realized that it is possible to apply the transforms to the Berlekamp-Massey algorithm itself, and Figure 5-13 presents a version of his time-domain decoder [10, Section 9.5], which can be extended to include both erasure handling and generator polynomials with $L \neq 0$. For our purposes, however, this vanilla version is sufficient.

There are n iterations in the main outer loop of the time-domain decoder, where n is the length of the codeword and must in general be a divisor of $2^m - 1$. The first $2t$ iterations comprise the Berlekamp-Massey algorithm cast back into the time domain, while the remaining $n - 2t$ iterations complete the error spectrum using (5.18), sequentially subtracting the contribution to the frequency spectrum from the codeword. Unfortunately, the frequency-domain vectors (i.e., error polynomials), which are of length $2t$ in the Berlekamp-Massey algorithm, are transformed to length n in the time domain; further, all n iterations are required before any corrected data can be output. Clearly the time-domain decoder has area-time complexity n^2, but Blahut has

```
/*        ASSUMPTIONS:                    */
/* 1. Codewords are multiples of g(x) = ∏_{k=1}^{2t}(x − α^k) . */
/* 2. The received word is s(x) = c(x) + e(x), with c(x) a codeword. */

   PROCEDURE TimeDomain;
   BEGIN                          /* initialize */
      FOR i := 0,1,...,n − 1 DO  BEGIN q_i := s_i; λ_i := b_i := 1 END;
      D  :=  0;
      FOR k  :=  1,2,...,n  DO
      BEGIN          /* compute output of latest filter */
          Δ  :=  ∑_{i=0}^{n−1} α^{ik} λ_i q_i ;
          IF k > 2t THEN          /* recursive extension */
              FOR i := 0,1,...,n − 1 DO q_i := q_i − Δα^{ik};
          ELSE                    /* compute error locator */
             IF Δ  =  0  THEN
                 FOR i := 0,1,...,n − 1 DO b_i := α^{−i} b_i;
             ELSE
                FOR i := 0,1,...,n − 1 DO t_i := λ_i − Δα^{−i} b_i;
                IF 2D  ≤  k − 1  THEN
                    D  :=  k − D;
                    FOR i := 0,1,...,n − 1 DO b_i := Δ^{−1} λ_i;
                ELSE
                    FOR i := 0,1,...,n − 1 DO b_i := α^{−i} b_i;
                ENDIF;
                FOR i := 0,1,...,n − 1 DO λ_i := t_i;
             ENDIF;
          ENDIF;
      END;
      FOR i := 0,1,...,n − 1 DO c_i := s_i − q_i;  /* correct errors */
   END;
```

Figure 5-13. Blahut's Time-Domain Decoding Algorithm

detailed a more complicated version [11] which also computes the error-evaluator polynomials and terminates with the Forney algorithm, thus reducing the complexity to order nt. Although the power-sum syndromes are never explicitly calculated, the computation of Δ in Figure 5-13 is extremely similar to syndrome generation but involves many more arithmetic operations. Thus, while it may be true that the time-domain decoder has a simple control structure, its area-time complexity certainly seems to limit its use to low-speed applications. In particular, unless the system has area on the order of n, a time-domain decoder cannot operate in full synchrony with the incoming data. The structural simplicity of the algorithm seems to lend itself more to sequential operation than to parallel computation.

5.9 Other Decoding Algorithms

Having been somewhat familiarized with the known Reed-Solomon decoding algorithms and their multitude of variations, one is led to speculate as to whether any other classes of decoders exist. There are many similarities between the known algorithms which could perhaps be generalized in an attempt to derive new decoders. For example, although the evaluation criterion varies, the update step is extremely similar in both form and purpose from algorithm to algorithm, and the evaluate-update cycle is repeated a maximum of $2t$ times, once for each component of error information. In fact, with the exception of Blahut's time-domain decoder, all the algorithms presented here have the same order of area-time complexity, namely t^2, although the associated constant varies slightly. Thus, one may conjecture that any decoding method will have the general control flow of the techniques already presented. As a case in point, a split remainder decoder (as discussed above) would be closely related to the known remainder algorithms and would likely bear a strong resemblance to the Berlekamp-Massey algorithm itself; it is therefore questionable whether such a decoder constitutes a separate class.

The known decoding algorithms can be roughly divided into two classes: power-sum syndrome (Berlekamp, Berlekamp-Massey, and Euclid) and remainder (Berlekamp-Welch, Liu) methods; again note that Blahut's time-domain decoder does not fit well into either category. Alternatively, these two approaches can be considered to operate in the frequency domain and the time domain, respectively. However, this viewpoint obscures an interesting relationship between the two approaches. The power-sum syndromes can also be regarded as dividing the received word by a linear polynomial $(x - \alpha^{L+j})$ to produce a constant remainder S_j; the remainder polynomial is the remainder produced when the received word is divided by the product of all these linear polynomials, $g(x)$. But, by the Chinese remainder theorem, any set of relatively prime polynomials whose product is $g(x)$ can be used to compute a set of remainders, which we will term generalized syndromes, with the same information content as any other set of such polynomials. Thus, there exists a family of decoding algorithms which operate on the generalized syndromes; the power-sum syndrome and remainder methods are but extremes in this family. For example, suppose

$$g(x) = \prod_{j=L}^{L+2t-1} (x - \alpha^j),$$

with $L = 2^{m-1} - t$ so that $g(x)$ is reversible. Then, if we let $\beta_j = \alpha^{L+j}$ and

$$g_j(x) = (x - \beta_j)(x - \beta_j^{-1}) = x^2 + (\beta + \beta^{-1})x + 1,$$

the set of generalized syndromes $\{ s_j(x) \equiv s(x) \pmod{g_j(x)} \mid j = 0, 1, \ldots, t-1 \}$ contains all the information necessary to decode. Observe that $g_j(x)$ can be generated by a two-stage encoder which involves only a single multiplier. Clearly there are many other such generalized syndromes which could be computed. Although the area-time complexity of the corresponding decoders would probably have the same order as the known methods, an understanding of this family of algorithms would illuminate the relationship between power-sum syndrome and remainder decoding. It is also possible that one of these generalized syndrome approaches would minimize the constant associated with the complexity order.

In fact, it would not be too surprising if all the known (and unknown) decoding algorithms are related by simple transformations. For example, the Berlekamp-Massey algorithm is a mild variant of the original Berlekamp key equation solver, and Blahut's time-domain decoder can be

formally related to the Berlekamp-Massey algorithm via Fourier transform arguments. Cheng [16] has also shown that the Euclidean method is formally equivalent to the Berlekamp algorithm, and Liu has made the first step in outlining the correspondence between remainder and power-sum syndrome decoding. There is room for much work in making these relationships rigorous, and it is clear that such results will provide a greater understanding of the decoding process.

Chapter 6

Bit-Serial Decoder Architectures

6.1 Partitioning the Decoder

In this chapter a new approach to building Reed-Solomon decoders is presented, in which all polynomial operations are done in a bit-serial, coefficient-parallel fashion, utilizing many finite-field arithmetic units. By contrast, conventional decoders perform computations in a bit-parallel, coefficient-serial manner with a single arithmetic unit. Obviously, the performance tradeoff between these two methods occurs roughly when the number of bits m equals the number of coefficients $2t$. Most of the codes of interest to us have $2t > m$, precisely because lower redundancy codes can be handled well by traditional decoders. It would be possible to employ multiple parallel arithmetic units, and such an approach merits further investigation for decoders of extremely high performance. However, bit-serial communication requires less overhead on a chip and still allows the construction of decoders with throughput higher than that of conventional decoders. Several decoding algorithms will be examined in light of this approach.

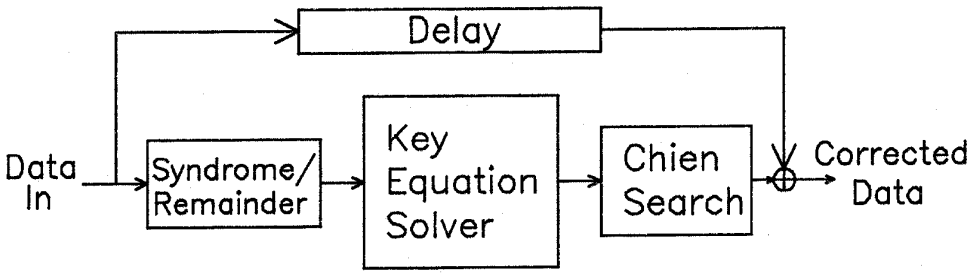

Figure 6-1. Decoder Block Diagram

The decoder can be divided into several basic blocks, as illustrated in Figure 6-1; we will not be concerned here with the delay line, which can be implemented using standard memory parts. It should be noted at this point that the major functional units of Figure 6-1 can all

operate concurrently. That is, while the first stage is computing the syndromes (or remainder) of the incoming word, the key equation unit processes syndromes from the previous block and the Chien search section calculates the corrections for the word received two blocks previously, as outlined roughly in Figure 6-2. Such parallelism allows the decoder to run synchronously with the incoming data, with a fixed latency from input to output, eliminating the need for complicated elastic buffers when dealing with a continuous stream of data blocks, and removing the penalty typically associated with worst-case performance. The important performance figure for such a decoder is the pipeline period P, which corresponds directly to throughput; e.g., a 10 MHz clock rate implies 10 Mbits/sec throughput. Because we are interested in utilizing all the available parallelism, each of the decoders which we shall consider has the basic architecture shown in Figure 6-1. It is to be understood implicitly that there will be (at least) one controller in the decoder to provide control signals at the appropriate time to each of the units.

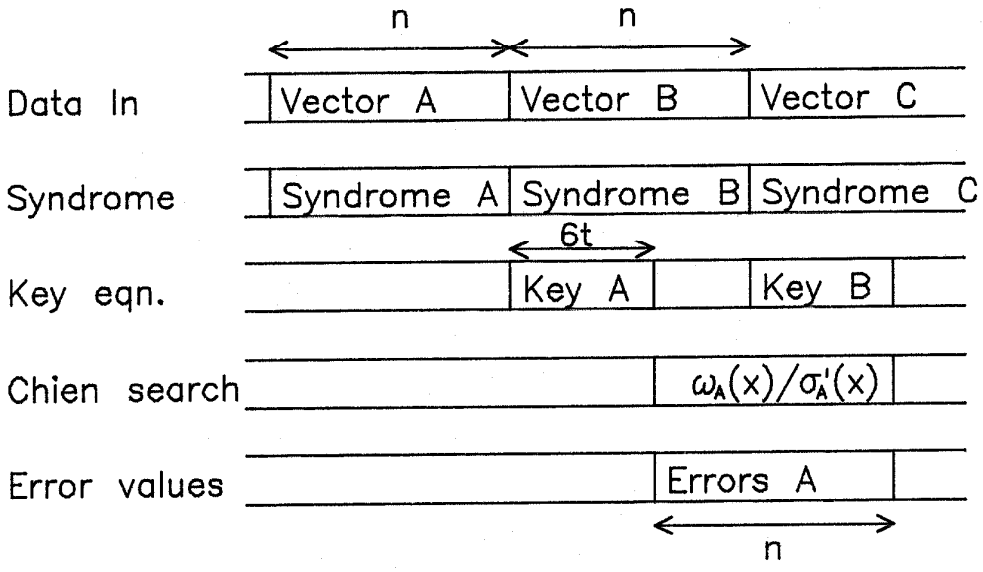

Figure 6-2. Decoder Timing Diagram

Let us examine each of the architectural blocks in turn before attempting to assemble the parts into a full decoder. First, we will explore the calculation of the syndrome/remainder from the received word, with a comparison of the features from these two basic approaches to decoding. Next, implementation of the Chien search and Forney algorithm in hardware will be discussed, since this step is common to all known algorithms. Structures for solving the key equation using several algorithms will then be presented and compared. Finally, consideration will be given to some applications and implications of a single-chip decoder.

6.2 Bit-Serial Syndrome and Remainder Computation

As discussed in previous chapters, the two major classes of decoding algorithms require a received word to be compressed into different (but entirely equivalent) forms for error correction to be applied. Power-sum syndromes comprise a portion of the discrete Fourier transform of the received vector, while the remainder upon division by the generator polynomial is a time-domain

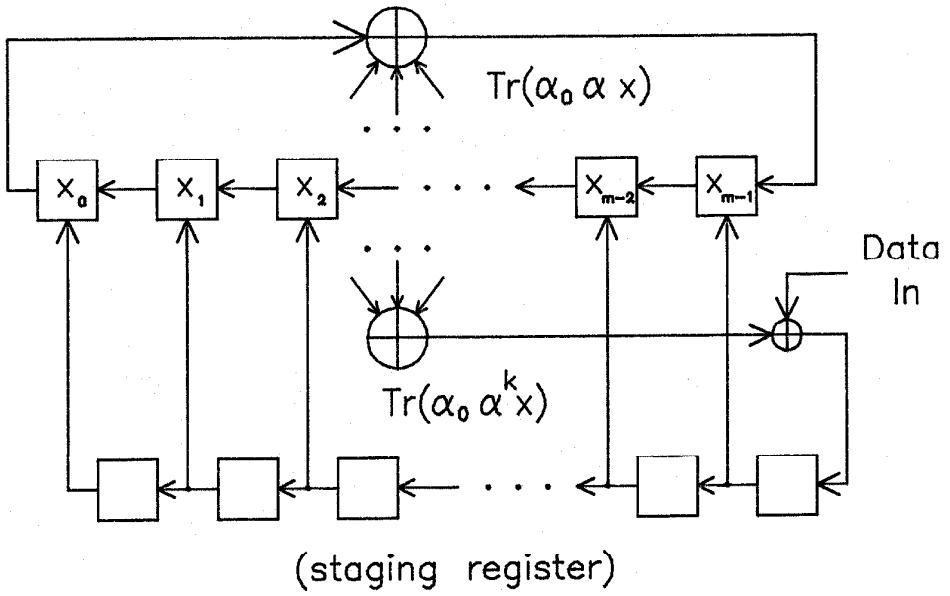

Figure 6-3. Dual-Basis Syndrome Computation

representation of the same information. The computation of either set of values can be efficiently accomplished in a bit-serial fashion. Let us illustrate the two cases in turn and then compare the results in view of the entire decoding process.

Perhaps the most straightforward implementation of power-sum syndrome computation can be derived from the polynomial viewpoint of the DFT. In other words, given a received vector $s(x)$, the syndrome $S_k = s(\alpha^k)$. As we saw in the first chapter, applying Horner's rule to the evaluation of S_k yields a simple recursive structure, in which the current value is multiplied by α^k and added to the incoming component s_i. A dual-basis implementation of this operation is outlined in Figure 6-3; observe that the multiplication by the constant α^k is hard-wired. Initially the x register must be loaded with zero. The bits of the product $x\alpha^k$ are added to the incoming data and shifted into the staging register; on every m^{th} clock, the x register is reloaded in parallel from the staging register. A similar structure exists using a shift-and-add multiplier (see Figure 6-4), but in a normal basis it is not possible to hard-wire the constant multiplication, implying the need for an additional register and many GF(2) multiplications. To compute the syndromes in parallel, $2t$ independent units can be constructed. If the clock is synchronized to the incoming data bits, the syndromes are calculated in real time; i.e., as soon as the last bit of the vector is received, the syndromes are available. Increasing the length of the staging register (in m-bit increments) allows the syndromes of interleaved words to be computed.

Given a received vector $s(x)$, the remainder polynomial, defined by

$$r(x) \equiv s(x) \pmod{g(x)},$$

where $\deg(s) < 2t$, can be computed by a linear feedback shift register, as illustrated in Figure 6-5. Since arithmetic takes place over GF(2^m), the feedback elements in general are not binary,

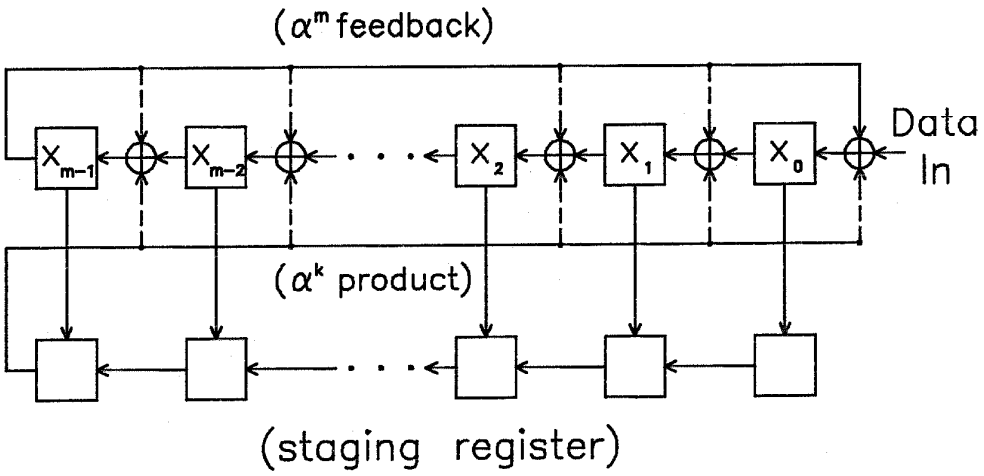

Figure 6-4. Shift-and-Add Syndrome Computation

but Berlekamp discovered an elegant method for computing the remainder $r(x)$ in a totally bit-serial fashion [5], using the dual basis. This approach is particularly effective when using a fixed code, because the multipliers for the coefficients g_i can be hard-wired. Another enhancement involves the use of *reversible* generator polynomials; i.e., selecting the parameter L of

$$g(x) = \prod_{k=L}^{L+2t-1} (x - \alpha^k)$$

such that the reciprocal of every root of $g(x)$ is also a root. It can then easily be shown that the coefficients satisfy $g_{2t-i} = g_i$, so that only half the number of field multiplications need be performed. If α is a primitive root of GF(2^m), then $L = 2^{m-1} - t$ will produce a reversible $g(x)$. A shift-and-add version of the same structure is fairly similar, but requires a parallel-load shift register instead of a strictly serial interface; again, when using a normal basis, the multipliers cannot be hard-wired.

In communication systems involving bidirectional information transfer, encoding is as important as decoding. One particularly attractive feature of remainder decoding is that the same structure which computes the remainder can also be used as an encoder; in fact, computing the remainder can be regarded as re-encoding the received word. However, in the following sections it will be shown that the key equation solver unit can often serve as an encoder for variable redundancy codes, so this apparent advantage of remainder decoding methods turns out to be not particularly significant. Further, there are several drawbacks to remainder decoding in hardware. For example, although bit-serial encoders can be built very efficiently for a fixed code, extending the structure to handle codes of varying redundancy modifies the coefficients g_i, thus requiring full (i.e., non-constant) multipliers. This change can significantly increase the amount of chip area dedicated to remainder computation, relative to the other major blocks of the decoder.

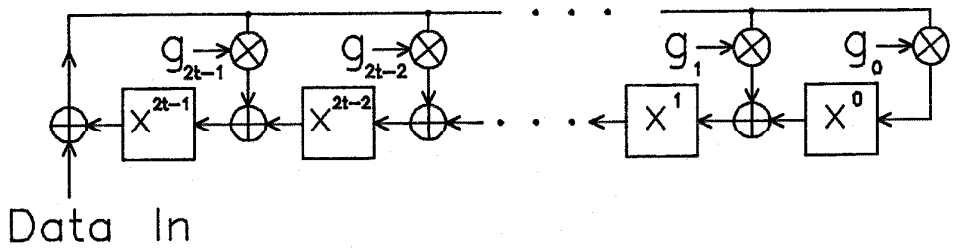

Figure 6-5. Remainder Computation

By contrast, a power-sum syndrome decoder can easily handle codes of varying redundancy by using only the syndromes needed. Suppose that $2t$ syndrome computation units are placed on the chip, sufficient to compute syndromes for a code C with generator polynomial $g(x)$. Then a code C' generated by $h(x)$ can be decoded if $h(x)$ is a divisor of $g(x)$; in other words, C is actually a subcode of C'. Only the syndromes corresponding to roots of $h(x)$ will be utilized by the key equation unit; the time cost of ignoring the additional syndromes is at most one m-bit cycle per syndrome to shift the field element out of a register while the key equation unit remains idle. The values of interest are a subset of the frequency window defined by $g(x)$, as shown in Figure 6-6. Since the same set of syndromes can be used for many codes, it is possible to handle varying redundancy using only constant multipliers, preserving the efficiency of the syndrome structures presented above. Also, if $g(x)$ is a reversible polynomial, a reversible $h(x)$ can be selected by centering its frequency window in the window of $g(x)$.

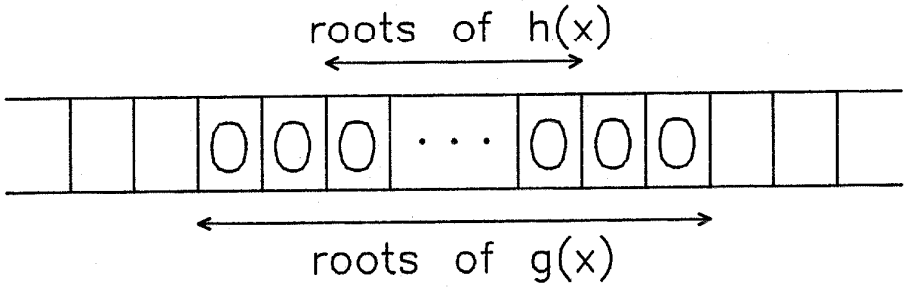

Figure 6-6. Subcode Frequency Window

The limitation of remainder methods with respect to varying redundancy, along with other similar problems encountered in the Chien search unit, leads us to conjecture that remainder decoding is best suited to a single-chip implementation if only a fixed code is desired. One of the initial goals of this research was to build a silicon compiler which could produce layout for a chip to decode a specific code. However, after some work it became apparent that, by introducing a few simple modifications, a power-sum syndrome decoder chip could handle all the codes of interest with little or no performance degradation. Thus, instead of having a compiler to generate a family of chips, it is possible to design a single chip for the entire family of codes.

Let us briefly analyze the area and time estimates for syndrome computation. The entire syndrome generator has $A = O(mt)$, since there will be $2t$ cells, each m bits wide. When using a shift-and-add multiplier, the feedback and product terms must drive on the order of m gates, so the clock period P would grow as $\log m$, assuming a logarithmic driver. For practical purposes, because we are only concerned with small m, this time is roughly constant; such fanout issues have been ignored by other authors [24] as well, since they are largely secondary. A dual-basis multiplier also involves a delay at least $O(\log m)$, depending on how the parity is structured. Observe that remainder computation involves the same area order but will generally run somewhat slower because of the need to distribute the products over the entire array of cells. Since m clocks are required to process each incoming component, the clock period could be interpreted as being proportional to m, especially if comparison is made with syndrome units which perform parallel multiplication. Usually, however, the figure of merit is the number of bits processed per second; thus we may say that P is essentially constant.

6.3 Chien Search

As we have seen previously, given the error-locator and error-evaluator polynomials from the key equation unit, in the case of power-sum syndromes, the error values can be computed using the Forney algorithm:

$$e_k = \begin{cases} 0, & \sigma(\alpha^{-k}) \neq 0; \\ \alpha^{-k(L-1)}\omega(\alpha^{-k})/\sigma'(\alpha^{-k}), & \sigma(\alpha^{-k}) = 0. \end{cases}$$

For remainder decoding methods,

$$e_k = \begin{cases} 0, & W(\alpha^k) \neq 0; \\ f(\alpha^k)N(\alpha^k)/W'(\alpha^k), & W(\alpha^k) = 0, \end{cases}$$

where the function $f(x)$ is related to the reciprocal of the generator polynomial $g(x)$, as described in Section 5.7. Because these two expressions are so similar, the basic strategy for error computation will be the same in either case. Evaluation of the factor $\alpha^{-k(L-1)}$ can be performed recursively using a single multiplier. However, the function $f(x)$ is considerably more complicated to implement; it could be very efficiently computed using a lookup table in ROM, but such an approach limits the chip to a fixed code. On the other hand, with the expression for $f(x)$ implemented in hardware, evaluation of $g(x)$ could be performed efficiently for a single code; unfortunately, much of this efficiency is also lost if varied redundancy is desired, since the multipliers for coefficients g_i cannot be hard-wired. Again we conclude that remainder methods are most effective for a fixed code. Thus, our examples in this section will concentrate on the power-sum syndrome case; extensions to the remainder methods are straightforward.

The goal of a Chien search is to find all roots of the error locator. Although efficient methods exist for finding roots of polynomials of degree four or less over fields of characteristic two [4, Chapter 11], each degree represents a special case, and we are still left with the general problem. As mentioned previously, while special cases can be very effective in software, they tend to be inefficient in hardware. Let us again make the assumption that, on each m-bit cycle, exactly one received vector component enters the chip and exactly one error value is output by the chip, with a constant latency between the input component and the corresponding output. In other words, as illustrated in Figure 6-2, the received word $s(x)$ arrives in the order

$$s_{n-1}, s_{n-2}, \ldots, s_2, s_1, s_0,$$

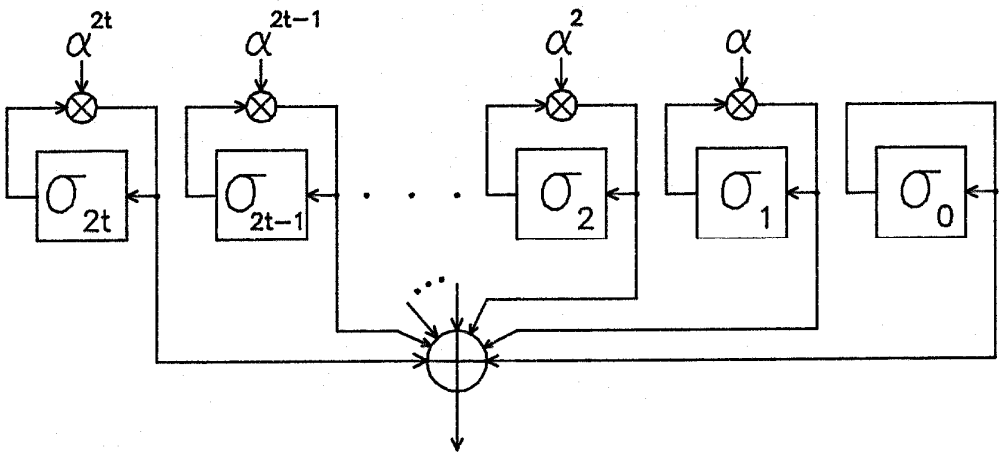

Figure 6-7. Chien Search Implementation

and the associated error pattern leaves the chip in the order

$$e_{n-1}, e_{n-2}, \ldots, e_2, e_1, e_0,$$

some fixed time later. Given this restriction, even if the roots and error values can be calculated in zero time, the decoder must wait for the appropriate time to output the error value. Implementing special cases for root finding given this assumption does not improve performance and thus it is difficult to justify the additional hardware cost. Also, if most of the corrupted codewords involve only one, two, or three errors, it could be argued that the channel is not being utilized effectively: the system design should take advantage of the available capability to correct more errors.

Fortunately, the Chien search provides a simple and general algorithm for finding all zeroes of a polynomial over a finite field [17]. Since there are only a finite number of possible roots of interest, one method is to try them all. Given the input ordering above, the appropriate sequence of values to compute in a Chien search is

$$\sigma(\alpha), \sigma(\alpha^2), \ldots, \sigma(\alpha^{-2}), \sigma(\alpha^{-1}), \sigma(1)\,. \tag{6.1}$$

Each zero in this sequence corresponds to an error, the value of which can be computed by the Forney algorithm. Consider the operator D introduced in Chapter five, defined by

$$Dp(x) = p(\alpha x).$$

Observe that application of D involves multiplying each coefficient p_i by α^i, so this transformation can be accomplished in parallel in hardware using only constant multipliers. If we define the sequence of polynomials

$$\sigma^{(k)}(x) = D^k\sigma(x) = \sigma(\alpha^k x),$$

then $\sigma(\alpha^k) = \sigma^{(k)}(1)$. Clearly $\sigma^{(0)}(x) = \sigma(x)$, and since $\alpha^n = 1$, it is also true that $\sigma^{(n)}(x) = \sigma(x)$. The sequence of Chien search values (6.1) is then given by

$$\sigma^{(1)}(1), \sigma^{(2)}(1), \ldots, \sigma^{(n-2)}(1), \sigma^{(n-1)}(1), \sigma^{(n)}(1)\,.$$

Evaluation of a polynomial at $x = 1$ involves summing up all the coefficients, so the error locations can be determined using a structure as shown in Figure 6-7.

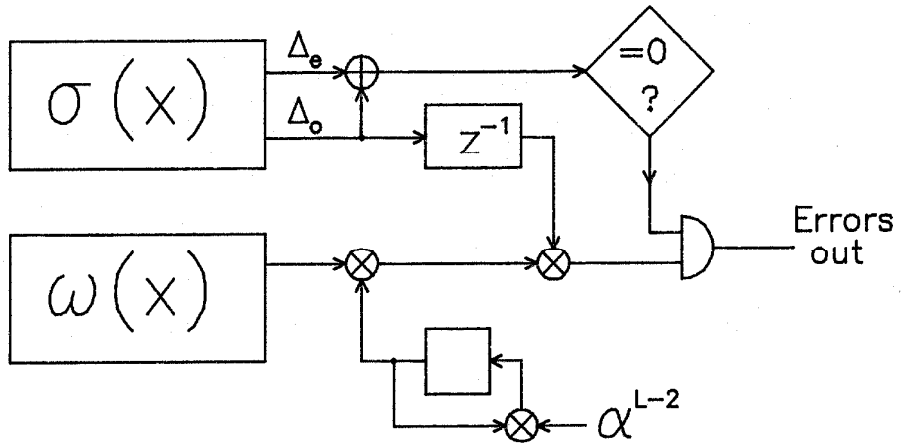

Figure 6-8. Forney Algorithm Using Chien Search Units

A unit identical to Figure 6-7 can also be used to evaluate $\omega(x)$ at the appropriate field values. Observe that, if the arithmetic operations are implemented in a bit-serial fashion, the large adder in the figure involves only $2t+1$ bits. Further, the clock period of this structure can be minimized by fully pipelining this sum, which adds a small constant to the latency without affecting the throughput. After each m bit clocks, values of $\sigma(\alpha^k)$ and $\omega(\alpha^k)$ are obtained for the next value of k. A central controller, obtaining this value from the end of the sum pipeline, can decide whether to apply the Forney correction based on the result $\sigma(\alpha^k)$. If this value is nonzero, the field element zero is output as the error component. Otherwise, it is necessary to compute $\omega(\alpha^k)/\sigma'(\alpha^k)$. At first glance this calculation seems to require another polynomial evaluation unit. Fortunately, polynomial derivatives simplify considerably when dealing with fields of characteristic two:

$$\sigma'(x) = \sum_{i=0}^{2t} i \bullet \sigma_i x^{i-1} = \sum_{\text{odd } i} \sigma_i x^{i-1}.$$

In other words, if we pipeline the summation in the Chien search polynomial unit so that all the even terms and the odd terms are added separately until the last stage, the derivative can be obtained from the odd terms. In fact, when $\sigma(\alpha^k) = 0$, the even and the odd parts are equal, so either partial sum will do. Given that we compute $\sigma(\alpha^k) = \sigma^{(k)}(1)$, the odd terms produce

$$\Delta_o = \sum_{\text{odd } i} \sigma_i^{(k)} = \sum_{\text{odd } i} \sigma_i \alpha^{ik} = \alpha^k \sum_{\text{odd } i} \sigma_i \alpha^{(i-1)k} = \alpha^k \sigma'(\alpha^k),$$

so the error value expression becomes

$$e_k = \alpha^{-k(L-2)} \omega(\alpha^{-k}) \Delta_o^{-1}.$$

The multiplicative inverse of Δ_o can be accomplished bit-serially, as discussed in Chapter four. A functional diagram of the entire Forney algorithm unit appears in Figure 6-8. Controls to

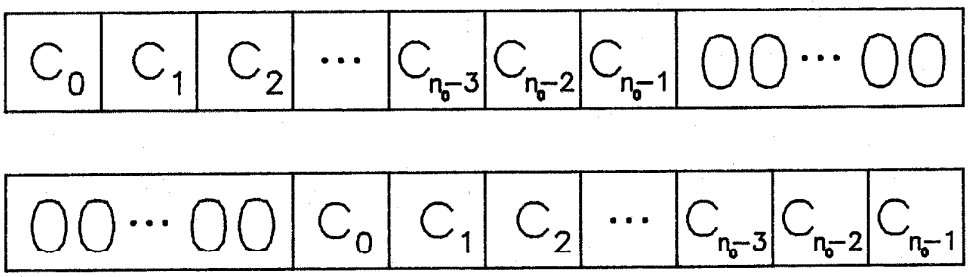

Figure 6-9. Possible Interpretations of Shortened Codes

load the appropriate initial value will come from a central controller on the chip, but with the registers properly initialized, the error pattern emerges sequentially from the output. Each block in Figure 6-8 can be implemented bit-serially, and pipelining can be added to decrease the clock period.

Before proceeding to the key equation unit, one additional limitation of remainder decoding should be explored. Often it is very desirable to use shortened codes, as explained in Chapter three, in order to match the code to a given block length. Suppose that the code is to have block length $n_0 < n = 2^m - 1$. Both the syndrome and remainder computations proceed as in the unshortened case, except that the calculation is terminated after only n_0 iterations, as one would expect. The problem arises in the Chien search: in order not to fall behind the next incoming word, all roots of the error locator must be found in only n_0 cycles. In the structures presented above, the root search begins at $x = \alpha$ and continues with sequential powers of α until all field elements have been tested. However, the only roots of interest here are at

$$x = \alpha^{n-n_0+1}, \alpha^{n-n_0+2}, \ldots, \alpha^{-2}, \alpha^{-1}, 1.$$

How can the additional $n - n_0$ elements be skipped? The answer to this dilemma is to utilize the cyclic properties of the code. Instead of considering the received word $s(x)$ to be

$$s(x) = \sum_{i=0}^{n_0-1} s_i x^i,$$

we may assume that

$$s(x) = \sum_{i=n-n_0+1}^{n-1} s_i x^i.$$

In other words, as shown in Figure 6-9, treat the shortened codeword as having trailing zeroes instead of leading zeroes. Because Reed-Solomon codes are cyclic, either interpretation of $s(x)$ is a codeword if no errors have occurred. Now the Chien search can be applied as before, stopping after n_0 corrections are computed.

However, this transformation has merely shifted the problem back to the syndrome (or remainder) unit. If we assume that the zeroes follow the received components, then the actual syndromes of interest are

$$\tilde{S}_k = S_k \alpha^{k(n-n_0)}. \tag{6.2}$$

Another way of deriving (6.2) is to invoke the translation properties of the Fourier transform, since we have effectively performed a circular translation of the time-domain values by $n - n_0$ positions. If the modified syndromes $\tilde{S}_k$ are used, both the key equation unit and the Chien search unit proceed as in an unshortened code, except that the decoding is terminated after n_0 components. Obviously, the key equation block has less time to complete the solution for σ and ω; we will see below that this processing time requires on the order of t component cycles, implying that there will be lower limit on the allowed rate of the code. In almost all the key equation structures we shall consider, the syndromes are introduced into the problem one at a time, so the modification (6.2) can be performed sequentially, as illustrated in Figure 6-10, given that the register is properly initialized to $\alpha^{L(n-n_0)}$. What about the remainder methods? Because the remainder is a time-domain quantity, the vector translation corresponds to a remainder translation; i.e., clocking the remainder unit $n - n_0$ more times with zero input, exactly as if the code were not shortened but had trailing zeroes. Unfortunately, there is no shortcut to perform this transformation, as in the syndrome case. One possibility would be to use a second remainder unit to perform the additional $n - n_0$ shifts, but this works only for $2n_0 \geq n$; in general we would require $\lceil \frac{n}{n_0} \rceil$ such units arranged in a pipelined fashion, introducing an additional latency of $n - n_0$ component cycles into the decoding process, as well as a large area overhead. Thus, it seems that remainder methods are inefficient at handling shortened codes in hardware.

Asymptotically, the area of the entire error evaluation unit will be dominated by the multiplicative inversion ROM, which grows as 2^m. However, for most m and t of interest, such a ROM will be considerably smaller than the polynomial evaluation units. Even in the case of large m, an alternative approach to reciprocal computation, when using a canonical basis, is to cast Euclid's algorithm into hardware, interpreting each field element as a polynomial modulo an irreducible polynomial of degree m over GF(2), as explained in Chapter two. This approach requires area proportional to m, thus allowing the construction of decoders over fields such as GF(2^{16}), which have typically been regarded as too large for practical purposes. Using this technique, we find $A = O(mt)$, since each of the $2t + 1$ cells have size $O(m)$. If we disallow the correcting of $2t$ erasures, valid error-locator polynomials are guaranteed to have degree less than $2t$, so only $2t$ cells would be required, exactly matching the number of syndrome units. Because the polynomial evaluation sum can be fully pipelined, all the parts have essentially constant pipeline period, with the possible exception of the reciprocal unit. Again, if needed, Euclid's algorithm can be employed in the reciprocal computation to reduce the time complexity to roughly constant order, so $P = O(1)$.

6.4 Key Equation Solution

We have now seen that the syndrome/remainder computation and the Chien search unit can be built fairly efficiently, with area on the order mt and roughly constant clock period. In fact, each of these blocks consists of $2t$ basically identical cells, differing only in the constant which is hard-wired into the multiplier. Observe that no full multipliers are required in either case. Also, the control signals are independent of the data being processed; values enter and exit at fixed times, regardless of the number of errors to be corrected. In fact, the only necessary controls are to tell each block to begin processing a new received word by either loading a multiplier register (in the case of syndromes) or loading a new coefficient (in the Chien search). These signals occur once per codeword; i.e., once every n component cycles, where a component cycle is defined to consist of the m bit clocks corresponding to a single character. These pulses to the two units differ slightly in phase, as shown in Figure 6-2, because of the time required to compute the error

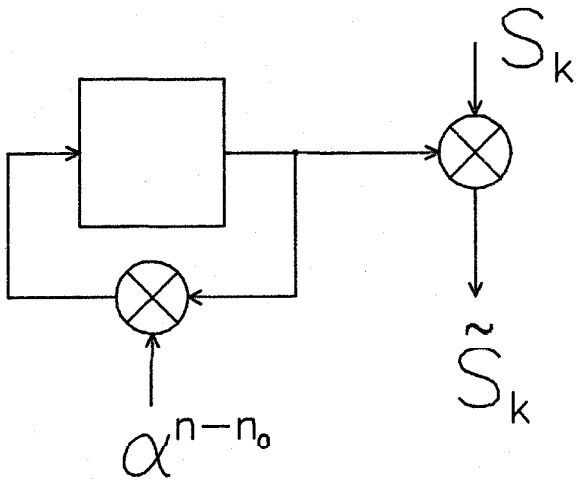

Figure 6-10. Syndrome Modification for Shortened Codes

polynomials from the syndromes. Although a fixed latency is always required in our architectures, it generally will be easy to tell that an incoming word was received correctly without completing the entire Chien search, by simply testing whether all the syndromes (or remainder coefficients) are zero. This test can even be performed sequentially, one syndrome at a time, introducing a latency of size $2t$, which will generally be small with respect to the block size n.

The final unit to be investigated in the block diagram of Figure 6-1 is the key equation solver. After presenting the original key equation solution method, Berlekamp outlined a parallel architecture for performing this algorithm, involving $t+1$ slave arithmetic units and a single master controller, and he observed that the decoding time was independent of t, although the hardware cost increased with t [4, Section 7.7]. At that time, actual construction of such a decoder was not cost-effective, but it is now possible to consider an implementation in VLSI. The architectures presented in the following sections will vary depending on the particular decoding algorithm chosen, but all are related in spirit to Berlekamp's original proposal.

Several features will be common to all the algorithms. First, like the syndrome and Chien search units, the key equation solver will consist of $2t$ cells, because the polynomials of interest involve at most that many coefficients. These cells will all be identical, but, unlike the previous blocks studied, each cell requires at least one full multiplier instead of a constant multiplier. As noted in the previous chapter, all of the decoding algorithms consist of $2t$ basic iterations, so the key equation unit will not always be busy, since a codeword consists of n component cycles. Each iteration, consisting of an evaluate and an update step, can require more than one component cycle; in general, the total processing time will be less than or equal to at for some integer a; in order to keep up with the incoming data, we must have $at \leq n$, or the code rate must satisfy

$$R = 1 - \frac{2t}{n} \geq 1 - \frac{2}{a}.$$

Clearly the addition of more cells in the key equation unit allows lower rate codes to be processed by the chip. One major difference between the key equation unit and the other blocks is that

the control signals here are dependent on the received data. Based on the result of an evaluate step, different update procedures must be followed, and signals to select the appropriate updated values must be sent to the cells. Fortunately, as we shall see below, the controller does not have to operate at the bit level; instead, it runs at the component level (m bit clocks), allowing fairly relaxed timing in the state machine circuitry.

6.5 Berlekamp-Massey Decoder

In this section we examine a key equation solver in greater depth than the decoders considered later, because many of the techniques presented here can easily be extended to other solution methods. A group of students at Caltech has spent months planning and laying out part of a decoder chip using the Berlekamp-Massey algorithm, providing greater insight into this particular architecture. A block diagram of the basic cell, which can be used to implement the version of the Berlekamp-Massey algorithm presented in Figure 5-8, is shown in Figure 6-11 for the case of dual-basis arithmetic. A similar structure exists using a canonical basis and shift-and-add multipliers. Each signal in the figure corresponds to a single bit line, except where noted by the usual convention at the parallel load input to the multiplier. If identical units are cascaded vertically, observe that inputs connect properly to outputs, so the design is quite regular. The floor plan requires $2t$ such cells to be stacked vertically, with adjacent syndrome computation and Chien search units, as illustrated in Figure 6-12. Note that the communication between these cells is entirely local.

In order to understand how this cell solves the key equation, the steps of the algorithm must be related to the signals and multiplexers shown in the figure. There are three main registers in the cell, one to hold the coefficient σ_i, one for a syndrome S_k, and one for the auxiliary coefficient τ_i. These registers are shown several characters long in order to accommodate interleaving, which will be necessary to utilize this structure efficiently, as we shall see below. Once the key equation is solved, the τ register will be used to hold the error-evaluator coefficient. Since $\tau(x)$ is never involved in field multiplications in this version of the algorithm, there is only one multiplier in each cell. Such an approach requires taking the reciprocal of the discrepancy Δ, but this operation can be performed bit-serially at a central controller. The alternatives are to build an additional multiplier at each stage or to multiplex a single multiplier in time, each of which significantly increases the area-time product, while utilizing an inversion ROM only adds a constant area to the controller.

Observe that the output of each multiplier goes to a sum over the entire coefficient array, for use in performing an inner product to calculate the discrepancy

$$\Delta = \sum_{i=0}^{k} \sigma_i S_{k-i}$$

during each evaluation phase. This same structure is later used in an identical fashion to produce the error-evaluator coefficients

$$\omega_k = \sum_{i=0}^{k} \sigma_i S_{k-i}.$$

Actually, the sum extends all the way up to $i = 2t - 1$, but, if we assume that $\sigma(x)$ is properly initialized so that $\sigma_i = 0$ for $i > 0$, then nonzero values of S in the staging registers of higher-order coefficients have no effect on the discrepancy computation. Since the degree of the error

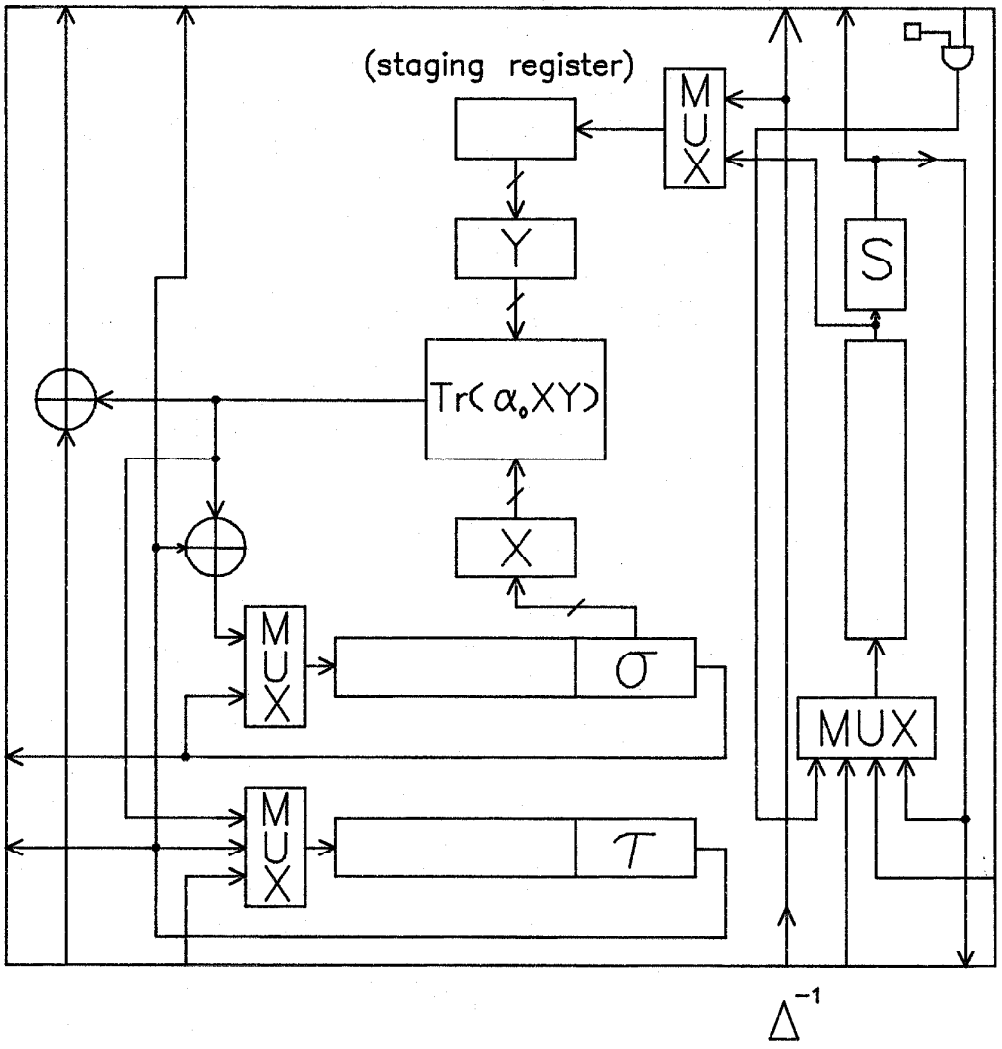

Figure 6-11. Berlekamp-Massey Key Equation Cell

locator grows by at most one each iteration, only one additional term is introduced into the sum Δ, as required by the algorithm. Similarly, during the computation of the error-evaluator coefficients, the syndromes are shifted down in the array. Clearly, since the sum extends all the way up to $i = 2t - 1$, zeroes should be introduced at the top, because σ_j cannot affect the value of ω_i if $j > i$. However, if a code of redundancy $2s < 2t$ is being decoded, zeroes must be introduced beginning at cell number $2s$. Thus, the upper right corner of the figure contains a one-bit register which performs a GF(2) multiplication (AND) to enable codes of varying redundancy to be handled. These control bits could be linked serially in a static shift register, to be initialized by the system controller.

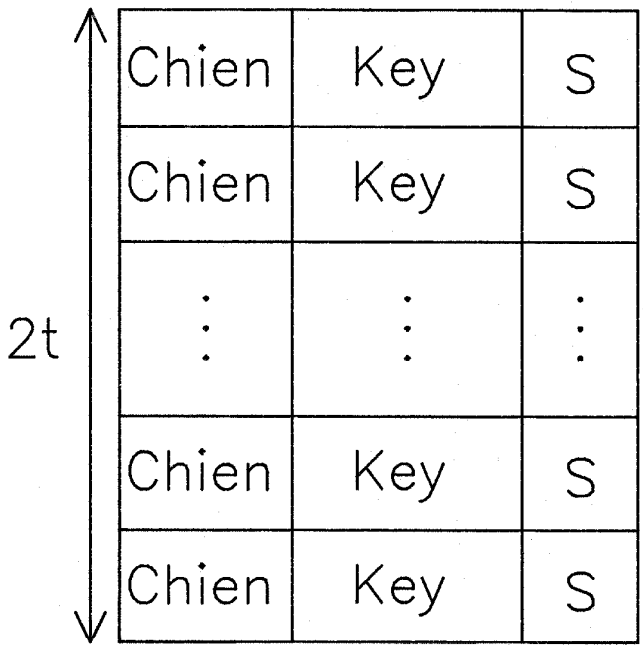

Figure 6-12. Decoder Cell Topology

Because the update step cannot be performed until the result Δ is known, there is a latency inherent in this sum, while in Euclid's algorithm there is no appreciable overhead in extracting the discrepancy term, which is just the leading coefficient of the current dividend. Citron therefore concluded that a Berlekamp-Massey key equation solver will always run slower than a Euclidean version in the same technology [18]. However, if interleaving is employed, as is usually desirable for greater burst protection, the Berlekamp-Massey discrepancy sum can be fully pipelined without affecting performance. In other words, while the value Δ proceeds through the sum pipeline and into the reciprocal unit, processing of the next interleaved character begins. A certain minimum depth of interleaving d is required if the entire pipeline time is not to affect throughput. Since the number of levels in the pipelined adder tree can be bounded by $\log 2t < m$, depth $d = 3$ seems to suffice when performing bit-serial arithmetic: one to load the staging register with the syndrome S_{k-i}, one to perform the multiplication, and a fraction ($\log 2t$ bit times) of a cycle to process the sum and begin outputting the reciprocal Δ^{-1} to the staging register again. Although the value Δ^{-1} is sent on a global line to each cell, a driver can be built in stages and pipelined to handle this large load, since there are roughly $m - \log 2t$ extra bit cycles available for this purpose.

The four multiplexers in the cell are used to control the operations to be performed during the evaluate and update steps. Fortunately, each multiplexer involves only individual bits. One of the conclusions of the student layout group was that wiring costs are already significant when using bit-serial techniques: changing to parallel communication would involve even greater overhead. A timing diagram showing the controls to each multiplexer for several

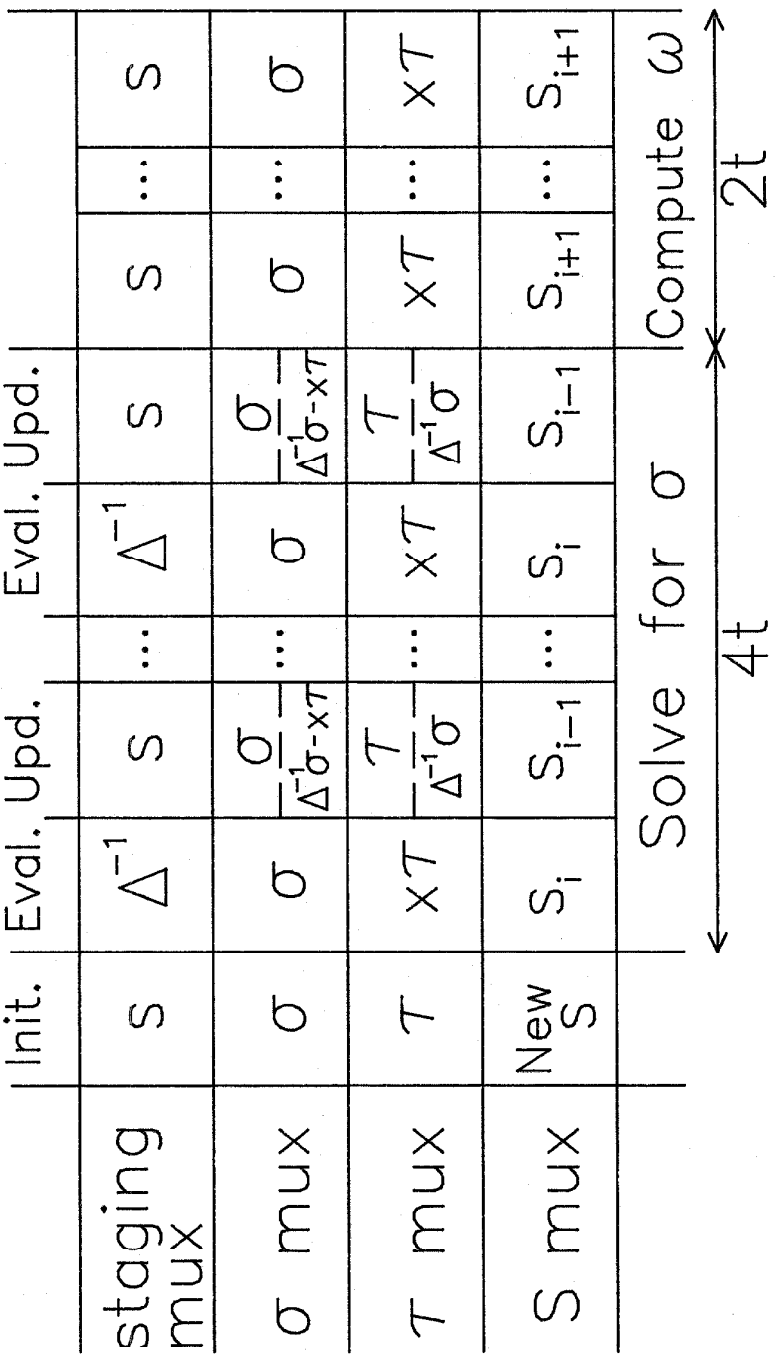

Figure 6-13. Berlekamp-Massey Controller Timing Diagram

evaluate-update steps of a typical key equation solution is shown in Figure 6-13. This figure does not take interleaving into account, so it should be interpreted only qualitatively. Some signals will actually be phased on different component cycles within an interleaved cycle. Let us explain the notation used by examining each multiplexer in turn. To compute the discrepancy during an evaluate phase, the staging register must have been loaded on the previous component cycle with a value from the syndrome register, which is an input to the staging register multiplexer. Once the discrepancy has been computed, the update phase requires multiplication of σ_i by Δ^{-1}, so the staging multiplexer must have been switched to this input on the previous cycle, loading the staging register with Δ^{-1}. Thus, the staging register multiplexer selects from one of two values, which are labeled Δ^{-1} or S in the figure. The auxiliary polynomial multiplexer selects between the current value τ, the shifted version $x\tau$, and the multiplier output $\Delta^{-1}\sigma$. During the evaluate phase, τ is shifted (multiplied by x) to be ready for the update step. From the algorithm, we see that the new value of τ is given either by $\Delta^{-1}\sigma$ or $x\tau$; since at this point the shift has already been performed, the multiplexer selects τ in the latter case. During the evaluate phase, σ remains unaltered, but an update selects either σ or $\Delta^{-1}\sigma - x\tau$, which is the other input to the error-locator multiplexer as shown, since τ has already been shifted. The syndrome register requires the largest multiplexer, with four inputs. Initially, the syndromes must be loaded from the syndrome unit, denoted in the timing diagram by 'New S'. During the evaluate step, the syndrome multiplexer selects S_i ; i.e., the current value is maintained in the register. While the update is occurring, S_{i-1} is selected, shifting the syndromes up to prepare for the next discrepancy calculation. The last input comes from S_{i+1}, shifting the syndromes in the opposite direction to produce successive coefficients of $\omega(x)$ once the error locator is known. Observe in Figure 5-8 that, corresponding to this ordering, the FOR loop index runs from $2t-1$ down to 0 . While these coefficients are being computed, they are fed into the τ_0 input at the bottom of the stack, and the τ multiplexer (which at this point could be relabeled ω) selects $x\tau$ to shift the coefficients up to their proper location in the array.

A few other features of the key equation cell should be mentioned. First, the syndromes are effectively introduced sequentially into the bottom of the array as the key equation solver proceeds. Thus, the syndrome modification for shortened codes, as illustrated in Figure 6-10, can easily be performed at the bottom of the cell array. Once the error polynomials are computed, they can be fed into the adjacent Chien search cell. For erasure handling, the controller must be able to store the erasure locations while an incoming word is being received. Then, the first h iterations proceed almost identically to the non-erasure case, except that the evaluate step is used to update τ to the new value of σ and in the update phase the erasure location is used instead of Δ. Observe that the entire key equation solution here requires roughly $6t$ coefficient cycles: $2t$ evaluate steps, $2t$ update steps, and $2t$ steps to compute $\omega(x)$, plus a few (two or three) additional cycles to initialize the registers to their proper values. A lower bound on the code rate using this decoder is thus given approximately by

$$R > 1 - \frac{2}{6} = \frac{2}{3},$$

which does not impose any significant practical limitations. For codes of higher rate, the key equation unit will be idle some fraction of the time. To utilize this available computational power effectively would require additional syndrome and Chien search units, multiplexing the key equation solver unit between problems. Such an approach can be attractive, particularly to

decode codes of very high rate at very high speed. Thus, there can be benefits to solving the key equation faster, but taking advantage of this capability involves a considerable area overhead.

As for the central controller, it is important to realize that each multiplexer control line will be steady for an entire coefficient cycle. The state machine period is m bit clocks, so the timing constraints are not particularly stringent. Besides providing signals to the key equation cells, the controller must send a pulse to the syndrome units to tell them to clear their accumulators and begin processing a new incoming vector, at exactly the same time when the key cells load the newly calculated syndromes. A similar pulse, phased roughly $6t$ coefficient cycles later, commands the Chien search array to load newly computed values of the error polynomials from the key equation units. The controller must also keep track of the integers D, h, and k of the Berlekamp-Massey algorithm (for each interleaved word). Based on the relationship between these parameters and the result of the test $\Delta = 0$, the controller sends out the appropriate signals to the σ and τ multiplexers during the update cycle. The integer arithmetic can also be done bit-serially, greatly simplifying the circuit design.

Before proceeding, it will be instructive to identify several features of the Berlekamp-Massey algorithm which make it attractive for implementation. First, a considerable reduction in area-time product is afforded by the fact that the algorithm is split. Observe that a similar architecture could be used with the original (non-split) Berlekamp algorithm, but such an approach would involve storing twice as many polynomial coefficients and either an additional multiplier or the multiplexing in time of a single multiplier. An appropriate figure of merit here would be the product of the number of coefficient cycles required to solve the equation and the number of multipliers plus the number of coefficients stored. For the Berlekamp algorithm, this figure is at least $32t^2$, while the Berlekamp-Massey figure is $24t^2$. Employing a split algorithm seems to be advantageous, but the Berlekamp-Massey superiority here seems to go even deeper. For example, such a gain does not seem readily available in the split Euclid algorithm, where the values σ_i must be computed recursively using the coefficients of $\omega(x)$:

$$\begin{aligned} \sigma_0 &= \omega_0/S_L \\ \sigma_1 &= (\omega_1 - \sigma_0 S_{L+1})/S_L \\ &\vdots \\ \sigma_j &= (\omega_j - \sum_{i=0}^{j-1} \sigma_i S_{L+j-i})/S_L. \end{aligned} \tag{6.3}$$

Because an inner product such as (6.3) is not needed in the gcd computations, this structure would have to be added, thus losing some of the efficiency of the Euclidean algorithm. On the other hand, in the Berlekamp-Massey procedure the same structure is used to calculate both the discrepancy terms and the coefficients of polynomial $\omega(x)$ given $\sigma(x)$, so this algorithm lends itself particularly well to being split.

Further, the Berlekamp-Massey solution of the key equation requires a fixed amount of time, regardless of the number of errors. By contrast, the Euclidean algorithm terminates more quickly if fewer errors have occurred. This feature has often been cited as favoring the gcd algorithm, because it would allow a key equation solver to be ready to accept a new decoding problem more frequently. However, to take advantage of such a capability, the Chien search unit would also have to find the roots of σ more quickly as well, but we saw in an earlier section that this problem is quite difficult. As mentioned above, one possible solution would be to have multiple

syndrome and Chien search units sharing a single key equation unit. But for our purposes, the fact that the solution requires exactly the same number of iterations each time greatly simplifies the design without adversely affecting the throughput. Another somewhat less important feature of the Berlekamp-Massey structure is that the polynomial coefficients stay in fixed locations; i.e., σ_0 will always be found in the bottom key cell. In Euclid's algorithm, where polynomials are continually shifted to align leading terms, the position (in space or time) of particular coefficients is not as certain, slightly increasing the complexity of aligning the coefficients with the corresponding Chien search cell. A seemingly related advantage is that no intermediate storage is required for the syndromes, which can be loaded in (coefficient) parallel from the syndrome computation unit without an intervening shift register, if the error polynomials are properly initialized as explained above. Finally, it is apparent that a Berlekamp-Massey chip, with only minor modifications, can be used to encode as well, since the facilities are available for polynomial shifting and distributing a common value to be multiplied by a distinct value in each cell and added to a local value.

There are nonetheless a few drawbacks to a parallel implementation of the Berlekamp-Massey algorithm. For example, since interleaving is not always appropriate, the necessity of interleaving using such a chip might be a problem. However, only the key equation unit requires interleaving, so if the code rate is sufficiently high, the key equation unit could keep up by performing a nop during what would otherwise be a cycle for an interleaved coefficient. The lower bound would then be $R > 1-(1/3)^2 = 8/9$, since the key unit is effectively utilized only one-third of the time. If the code rate satisfies this constraint, the Berlekamp-Massey structure would be acceptable. Another problem is that the architecture is non-systolic, making it somewhat more difficult to distribute the decoder over a large chip or several chips, particularly in view of the need to compute the inner product. However, even this difficulty can be overcome by appropriate use of pipelining, possibly introducing the need for additional levels of interleaving.

There are two contributions to the area of the key equation section. The first involves the multipliers and shift-registers shown in the block diagram of Figure 6-12, and the second involves the additional storage required to handle interleaving to depth d. The size of the key equation cell is roughly proportional to m, but the interleaving memory has size $(d-1)m$, so the overall area per cell goes as md. Thus, the entire key equation array has area $A = O(dmt)$. Although the depth of interleaving has been ignored in analyzing the area estimates for the syndromes and the Chien search blocks, it is readily seen that in fact these areas scale in exactly the same way. As for clock period, we have already seen that the global communication required in the key equation unit can be pipelined, so the delay is roughly determined by the multiplier delay $\log m$; again, since we are not interested in asymptotic m, this figure can be regarded as roughly constant. However, if part of the interleaving storage is implemented using a RAM, as was the case in the student chip project, the memory cycle time may become the dominant factor. For a RAM shift register which is dm bits wide, the access time will increase roughly as dm, but the proportionality factor will be fairly small with respect to the constant overhead time—if clever circuit design is used. Also, it is possible to access several bits in parallel, taking the RAM out of the critical timing path. In other words, the limiting consideration in clock period, to first order, may well be the distribution of the clock signals to the $2t$ coefficient cells. Thus, for our purposes, we again have $P = O(1)$.

6.6 VLSI Implementation of Berlekamp-Massey Decoder

A group of four students (Bob Anderson, Anne-Marie Brest, Neil Brock, and Brenda Roder) from the VLSI design class at Caltech worked on implementing a prototype Berlekamp-Massey decoder

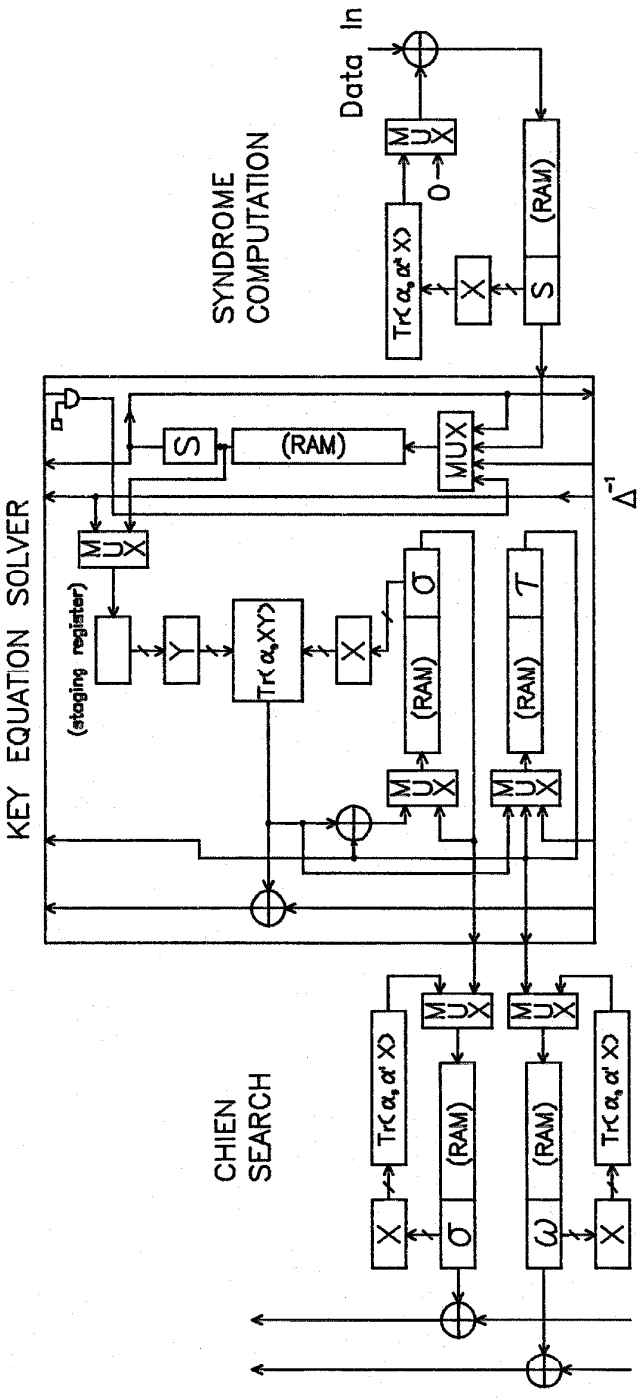

Figure 6-14. Berlekamp-Massey Coefficient Cell

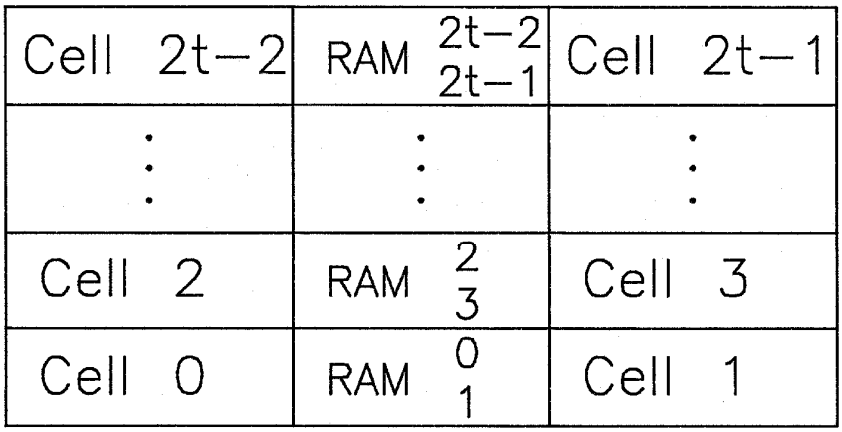

Figure 6-15. Decoder Floor Plan

chip in NMOS using Mead-Conway design rules [21] as a class project for the winter and spring academic quarters of 1984. The main goal of this project was not to produce a working chip, since the amount of time each student could dedicate to the design was limited. Instead, it was hoped that the major cell designs (e.g., finite-field multipliers) could be completed and a floor plan specified in order to get a good estimate for the total area of a decoder chip, and that a detailed analysis of the timing, necessary to specify the controller state machine, would allow any architectural problems to be discovered.

A full block diagram for a single coefficient cell, including the syndrome, key equation, and Chien search units, is shown in Figure 6-14. Six shift registers in the figure consist of multiples of m bits to allow for interleaving, and $m = 8$ was chosen for this design. To conserve both area and power, most of this storage was laid out using a three-transistor dynamic RAM array configured as a shift register, although exactly m bits are implemented using conventional shift registers because of the need to perform parallel transfers to the multiplier units. The controls for this RAM can be shared between all the coefficients, and only a single read/write line is required per bit: while one bit cell is being read, the adjacent cell is written. A shift register with accompanying bootstrap drivers is used to step the control lines through consecutive bit cells, eliminating the need for an address counter and decoder. Sections of the RAM can easily be bypassed, thus allowing the decoder to handle varying degrees of interleaving. A prototype RAM chip has been sent out for fabrication, consisting of 48 rows (enough for eight coefficient cells) of 32 bits each. Such a memory would allow interleaving of up to depth five over GF(256). The dynamic RAM cell is roughly 17λ by 21λ, so the RAM storage for each coefficient cell is about 600λ wide and 130λ high.

Figure 6-15 illustrates the basic floor plan for the decoder chip. Initial estimates of the coefficient cell size showed a large discrepancy in height, so it was decided to interleave rows of RAM and place cells on either side as shown, effectively doubling the height of the RAM array. Computing the derivative of $\sigma(x)$ during the Chien search requires separate sums over the even and odd terms, so the even and odd coefficients were placed on opposite sides of the RAM, facilitating the separation of the sum. Consecutively numbered cells must be able to talk to each other, but fortunately outputs from the RAM are available at the bit line on either side, so the

communication paths already exist. In retrospect, this physical separation of even and odd terms may not be necessary, since it would not be particularly difficult to build two distinct adder trees on one side. However, interleaving the rows of the RAM by placing coefficient cells on either side does appear to allow a fairly good match between the coefficient cell height and the RAM size.

The bit-serial multiplier was constructed using the dual-basis technique with the primitive polynomial $f(x) = x^8 + x^4 + x^3 + x^2 + 1$. A single cell is roughly 70λ wide and 180λ high, including two staging registers and the x and y registers. Thus, a full multiplier over GF(256) is 560λ by 180λ, only slightly larger than initial estimates. The feedback term for x can be either included or bypassed in this cell; similarly, it is simple to remove the y staging and holding registers and hard-wire a constant value for y. The cell is drawn in Figure 6-16, with the feedback term present. Observe that the value y is stored dynamically in the multiplier. For speed purposes, the parity chain is fully precharged on one phase of the clock, then discharged on the other phase while the data are held stable. It is clear that some area optimization could be performed, but the design is largely metal limited in the vertical direction. In particular, the parity chain required in the dual-basis structure seems to be somewhat inefficient in area; a second layer of metal or polysilicon wiring would help tremendously here. As a result, perhaps a shift-and-add multiplier structure can be laid out in considerably less area than a dual-basis multiplier in this technology.

A coefficient cell consists of three constant multipliers and one full multiplier, wired together with several XOR gates and small multiplexers. Although each of the leaf cells has been designed, the wiring of all the parts together has only been completed topologically on paper. The total area for this structure was then estimated to be roughly 400λ high and 1500λ wide. Thus, including RAM and coefficient cells, the total size for the decoder, which is capable of handling redundancies up to sixteen over GF(256), is 3200λ high and 3600λ wide. With $\lambda =$ 1.5 microns (i.e., 3μ feature size), this portion of chip measures roughly 5mm on a side. To this section must be added the controller, consisting of a fairly small state machine with some bit-serial integer arithmetic units, two inversion ROMs (one for the key equation and one for the Chien search), and the final stages for computing the error values, as shown in Figure 6-8. Although the controller has not been fully built, the reciprocal table has been designed, and the ROM array is smaller than the LFSR which generates addresses for it. Clearly the controller will be considerably smaller than the rest of the chip; a conservative estimate is that roughly fifteen percent of the chip area would be dedicated to the controller.

Thus, it is quite possible to fit such a decoder on a single chip using today's technology. As feature sizes shrink, even larger redundancy can be handled. With regard to speed, the major delays (which cannot be pipelined) would seem to be the parity chain in the multiplier, the bit-serial inversion ROM, and the storage RAM. Careful use of precharging should allow even these times to be minimized, and it is expected that the chip can run easily at a 10 MHz clock rate, corresponding to a 10 Mbit/sec throughput. If close attention is given to clock and other driver circuitry, it should be possible to achieve double the speed using existing MOS technology.

6.7 Euclidean Decoders

In this section, three architectures for implementing a key equation solver using the gcd algorithm are presented, and it is intriguing that this approach lends itself so well to different computational structures. Perhaps the reason is that the discrepancy term in Euclid's algorithm consists only of the leading coefficient of a polynomial, which can be quite easily extracted in a wide range of architectures. By contrast, the Berlekamp algorithm requires an inner product and remainder

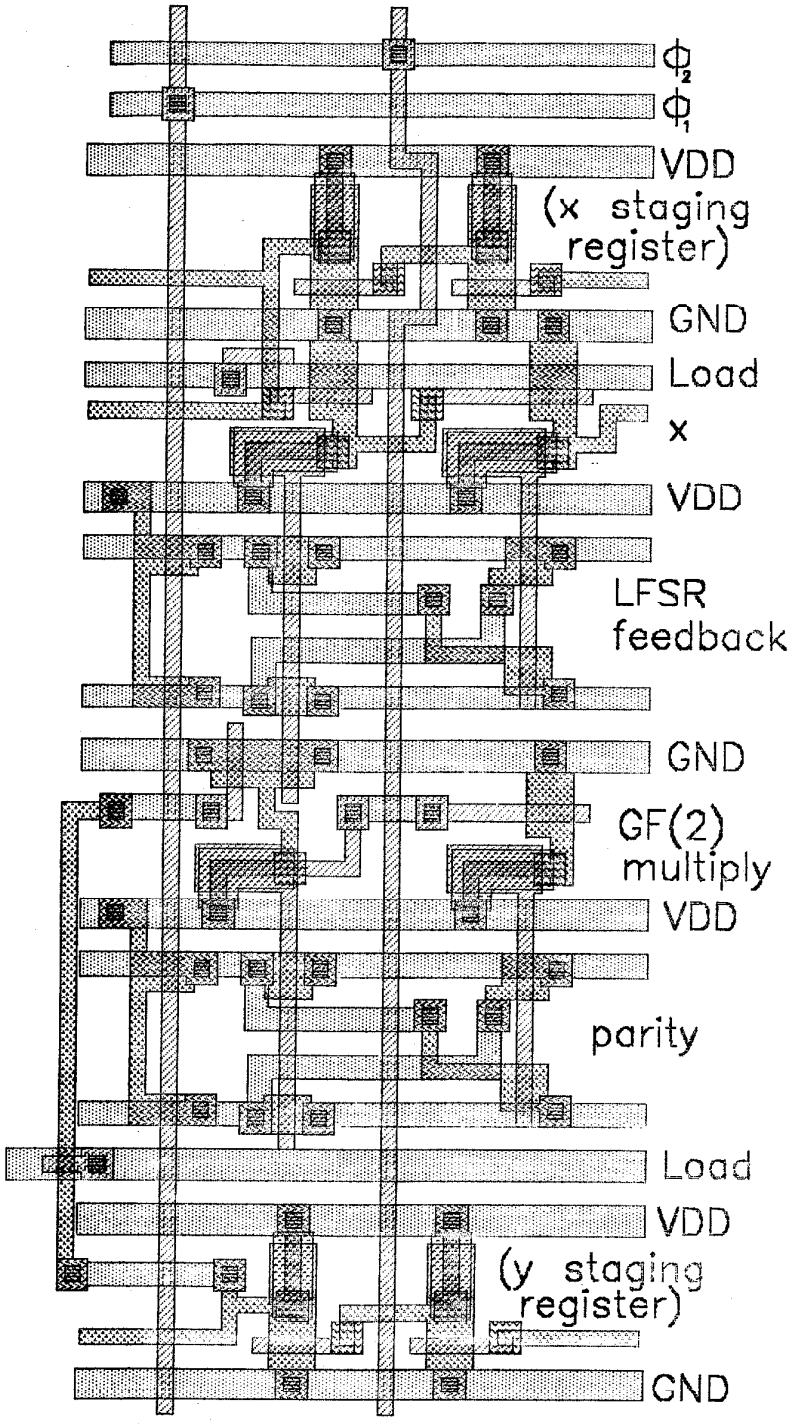

Figure 6-16. **One Bit Slice of Dual-Basis Multiplier**

methods involve polynomial evaluation; apparently, the complexity of the evaluation criterion in these two cases strongly dictates the hardware configuration. Hopefully, the simplicity of the Euclidean discrepancy term will also allow the key equation unit to be utilized efficiently without the requirement of interleaving.

As mentioned previously, using a split approach to the gcd computation does not seem as attractive as in the Berlekamp algorithm, because to compute the error locator from the error evaluator requires performing a series of inner products, and such a structure is not otherwise needed. Unfortunately, if an inversionless version of the non-split algorithm is employed, four multiplications must be performed per coefficient cell per evaluate-update step, as opposed to an equivalent of three for the Berlekamp-Massey algorithm. Further, the storage for two additional polynomials and the need for the extra multiplications (in either time or space) dictate that fewer cells will fit on a chip, reducing the amount of redundancy which can be handled. Fortunately, however, it may be possible to exploit other properties of the gcd procedure to effectively halve the amount of computation. It can be shown [40, Section 8.4] that, at each step of the algorithm, the sum of the degrees of $\omega(x)$ and $\sigma(x)$ is exactly $2t - 1$, in the case of no erasures. In other words, while the degree of $\omega(x)$ shrinks throughout the procedure, the degree of $\sigma(x)$ increases accordingly. Because identical operations are applied to the two sets of polynomials, a decoder could take advantage of this property by allowing σ to grow into registers being vacated by ω. Unfortunately, if erasure decoding is included, the sum of the degrees is always $2t + h - 1$, so roughly twice the number cells is again required. Berlekamp introduced a structure for implementing this technique [4, Section 2.3], requiring some additional control logic at each stage to separate the two sets of polynomials. So, Euclid's algorithm in hardware also seems to split naturally, although the term has a slightly different meaning than in the Berlekamp-Massey case. The important fact is that an errors-only decoder may be able to operate without a performance penalty using roughly half the area which one would originally estimate.

Euclid's algorithm consists of a very simple sequence of operations, as outlined in Figure 5-9. Given two polynomials $\omega(x)$ and $r(x)$, with $\deg(\omega(x)) \leq \deg(r(x))$, a shifted and scaled version of $\omega(x)$ is added to $r(x)$ so as to cancel its leading coefficient, thus decreasing the degree of $r(x)$ by at least one. This process continues until the degree of $r(x)$ drops below that of $\omega(x)$, at which point the roles of the two polynomials are exchanged and the process continues. For decoding purposes, the algorithm terminates when the degree of $\omega(x)$ drops below some value, typically t in the errors-only case. Data flow is determined entirely by these remainders; the accompanying polynomials $v(x)$ and $\sigma(x)$ are handled identically to the remainders but have no say in the matter.

Our first proposed architecture implements this procedure very literally and has a floor plan similar to the Berlekamp-Massey approach, as shown in Figure 6-17. Each cell stores the four coefficients ω_i, r_j, σ_k, and v_l, and has the ability to perform the following update operations:

$$r_j := r_j - p\omega_i \qquad \text{and} \qquad v_l := v_l - p\sigma_k,$$

where p is the ratio of the leading coefficients of $r(x)$ and $\omega(x)$. Linear scaling transformations can be applied to remove the need for the reciprocal in this ratio, but such a method doubles the number of multiplies per cell instead of requiring one inversion and multiply at the central controller; the appropriate tradeoff depends on the application. Each cell has the ability to exchange pairs of polynomials and shift $r(x)$ and $v(x)$ coefficients up one stage to align leading terms. It is important to realize that, at any given time, the position in the array of a particular

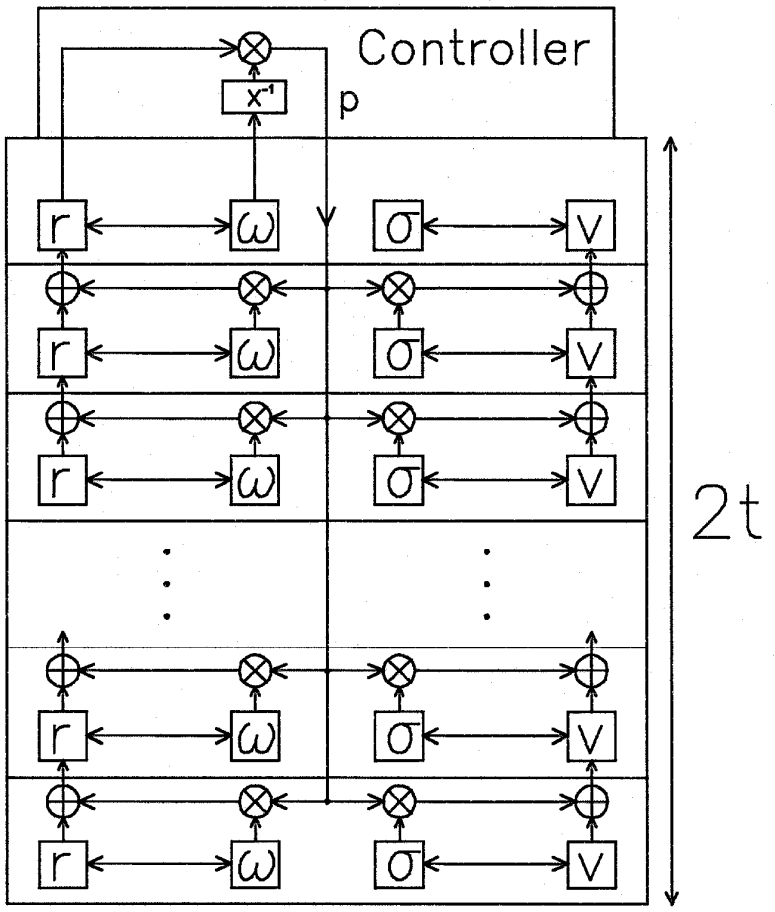

Figure 6-17. Non-Systolic Euclidean Key Equation Solver

coefficient, say σ_i, depends on the error pattern, although coefficients σ_i and σ_{i+1} are always in adjacent cells. Leading terms of $r(x)$ and $\omega(x)$ are always aligned at the topmost cell, so that the discrepancy is immediately available to the controller. Control signals to the array in a typical situation would go something like: shift, scale by p_i, add, shift, scale by p_{i+1}, add, swap, shift, scale by p_{i+2}, add, etc. The controller keeps track of the degree of the two remainder polynomials, stopping at the appropriate time. Because the final coefficients will be top justified instead of bottom justified in the array, some care must be taken to load them into the Chien search unit correctly. Also, since initially the syndromes are all required at once, either the modification for shortened codes must be performed in parallel or an additional latency will be introduced while the syndromes are modified sequentially. The array is clearly non-systolic, so it would be difficult to distribute the structure over several chips, but it is clear that encoding can be accomplished using Figure 6-17. This architecture will not be examined in further detail here because of its similarity to the Berlekamp-Massey approach and its intuitive simplicity.

A second implementation strategy for Euclid's algorithm was suggested by Kung et al.

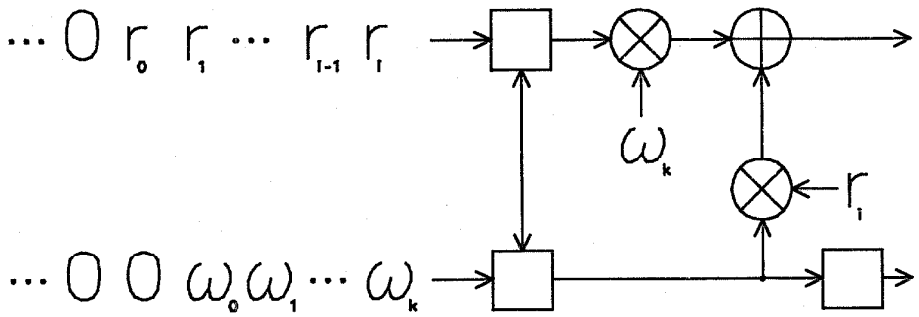

Figure 6-18. Systolic GCD Cell

[14], in which polynomials are considered to be a stream of coefficients, high-order first, padded at the end with a sufficient number of zeroes. If the leading terms are aligned, observe that a cell such as Figure 6-18 will perform one atomic step of the gcd procedure, decreasing the degree of $r(x)$ by at least one. As the leading coefficients enter the cell, they are latched to be used in the multipliers on all further cycles. Identical transformations are applied to the polynomials $\sigma(x)$ and $v(x)$. In fact, if the initial values of these polynomials, padded with appropriate leading zeroes to comprise $2t$ total coefficients, are appended to the corresponding remainder stream, the array will compute both the error evaluator and error locator in turn, effectively splitting the algorithm. The controls are set up to insure that the leading coefficient of $\omega(x)$ is always nonzero. Thus, if the leading term of $r(x)$ is zero, the net effect of the cell is to delay $\omega(x)$ with respect to $r(x)$, aligning the next coefficient of $r(x)$ with $\omega(x)$ for input to the next cell. Eventually the leading nonzero coefficient of $r(x)$ will catch up. Observe also that in this case both $r(x)$ and $v(x)$ are multiplied by the same scalar, so the final ratio of error polynomials is not affected. At each cell, if the leading term of $r(x)$ is nonzero and if the degree of $r(x)$ drops below that of $\omega(x)$, the polynomials exchange roles and the process continues. Since each stage decreases the degree of $r(x)$ by at least one, exactly $2t$ such cells are needed to insure that the key equation solution will be completed. When the degree of $r(x)$ drops below t, all further stages just pass the polynomials through unaltered, and the error locator and error evaluator emerge from the end of the pipeline. A group at the Jet Propulsion Laboratory has begun the design of a decoder over GF(16) based around this architecture [32], using normal-basis multiplication.

The systolic nature of Kung's architecture provides several benefits. First, there is no need to drive signals (other than clocks) across the entire chip. Second, a fixed time is required to solve the key equation, simplifying the interface to a Chien search unit. Also, the key equation unit is fully available to begin solving a new problem after all the coefficients have been input, if some effective way can be devised to utilize the error polynomials output at this rate. Further, if matched delays are inserted on all paths between any two adjacent cells, the only impact is a corresponding additional latency through the array. Thus, for example, it would be possible to distribute the cells across several chips, perhaps converting from bit-serial to multi-bit communication on the pads, without suffering a throughput penalty. This feature is extremely attractive, because it allows the decoder to handle codes with higher redundancy than could be accommodated on a single chip. Lastly, because the polynomials enter serially into the array, syndrome modification for shortened codes can be performed easily just before the first cell.

However, Kung's approach also has several drawbacks. For example, the structure does

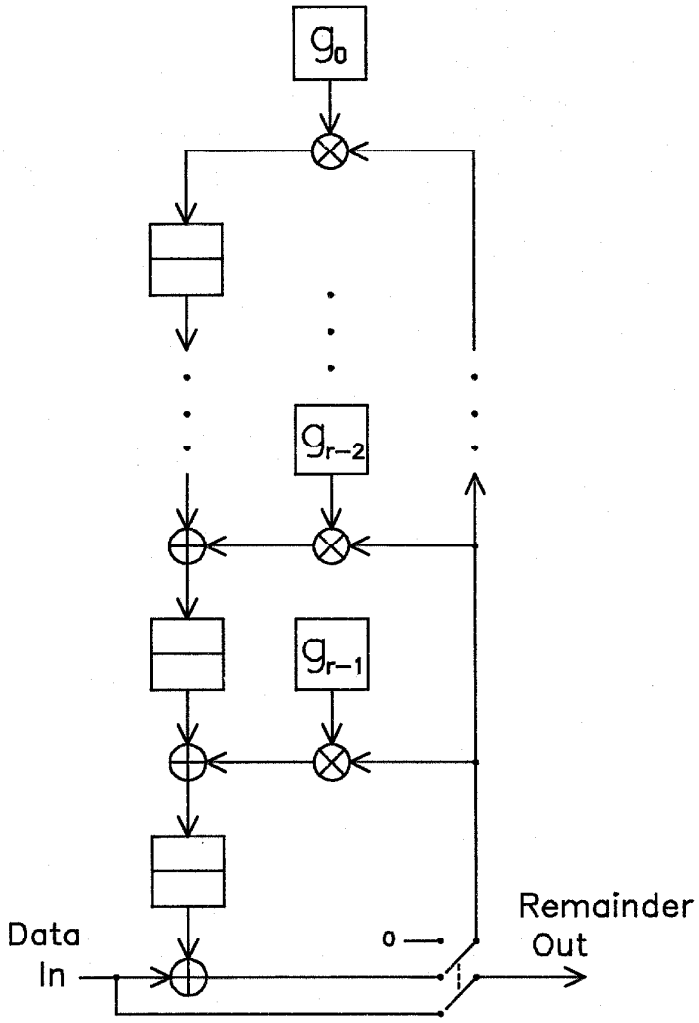

Figure 6-19. Non-Systolic Interleaved Encoder

not seem to lend itself to encoding, implying the need for a separate encoder block. Perhaps the most serious problem is that each cell in the gcd array requires a separate controller to keep track of the polynomial degrees, determining when to exchange the roles of the two polynomial pairs and when to terminate the algorithm. It is likely that such a controller will be comparable in area to the rest of the cell, so an enormous area penalty is paid with such a design. Another consequence is that it is not feasible to consider a version of Euclid's algorithm which involves multiplicative inversion. Such an approach would reduce the number of multiplies in the cell; unfortunately, this gain is nullified by the cost of a reciprocal ROM and a multiplier required in each controller. The inversionless algorithm requires twice the number of multiplies per cell as in the non-systolic key equation unit presented above. It does seem that this architecture has

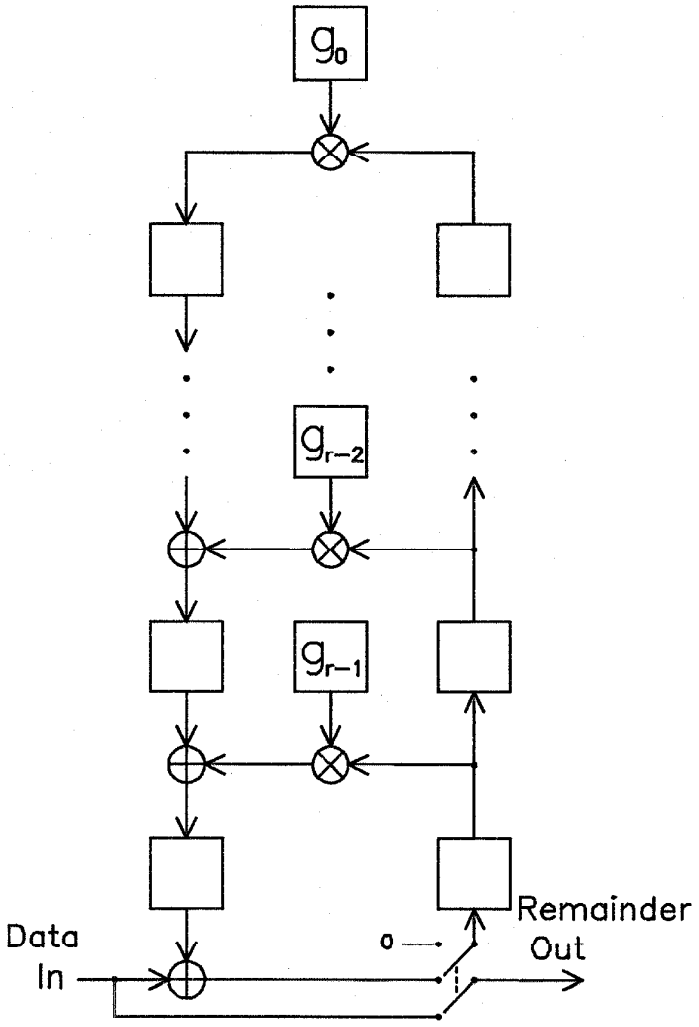

Figure 6-20. Systolic Interleaved Encoder

an important application in bit-serial reciprocal computation over fields GF(2^m) which are too large for table lookup. Using a canonical basis with irreducible polynomial $f(x)$, the coefficients f_i and the components of the element x to be inverted are fed into the array, and the reciprocal emerges from the other end. Because the bits of x are only required serially, this structure seems to fit in well with the strategy of a bit-serial decoder scheme. Exactly $2m$ cells would be required to complete Euclid's algorithm.

Clearly, there are area advantages to having only a single central controller, but significant improvements in performance and flexibility are provided by a systolic design. A decoder combining these two features would thus be extremely attractive, and Dally and Whiting at Caltech have recently proposed such a structure [23]. To motivate this architecture, let us

first consider the problem of building a systolic encoder. In other words, given the generator polynomial $g(x)$ of degree $r = 2t$, with $g_r = 1$, we wish to take the remainder of the received vector $s(x)$ mod $g(x)$. Since Euclid's algorithm consists of repetitively taking one polynomial modulo another, an encoder should point the way to a decoder architecture. A conventional encoder circuit was shown in Figure 6-5, in which the incoming value is added to the output of the high-order coefficient and fed back to be multiplied by the generator polynomial and added into the shift register. This structure is obviously not systolic, since the feedback term must be distributed to all the multiplier inputs.

A first intuitive attempt at making the array systolic might be to add delay elements at each stage along the feedback, but this change does not produce the desired effect, since the loop delays are not preserved. That is, for example, the feedback term must be able to reach the final adder in only one cycle after going through the g_{r-1} multiplier, as this is clearly the case in Figure 6-5. With the additional delay stages, two cycles are required for this to occur; in fact, all loop delays are doubled. However, consider the fact that both the loop delays and the amount of storage are doubled. This phenomenon leads us to suspect that perhaps some interleaving is present in our new structure. To see that this is indeed the case, consider Figure 6-19, which presents a conventional (non-systolic) encoder, interleaved to depth two. With the array initially set to zero, the $n - r$ high-order coefficients (from both codewords) are entered with the feedback switch on. During the remaining r interleaved cycles, the feedback is turned off, clearing the array and completing the remainder computation. A new pair of codewords can then be entered. If we now introduce delay elements along the feedback path, all loop delays can be preserved by removing one delay from each segment on the left, producing the systolic structure of Figure 6-20, which is functionally identical to the non-systolic encoder of Figure 6-19. At any time, vertically adjacent registers hold values from alternate encoding processes. Observe that this architecture can handle codes of variable redundancy by setting the uppermost generator coefficients to zero, effectively removing the upper portion of the array from the computation. Also, various depths of interleaving can be accommodated by adding delay elements; in the systolic case, the delay can be added on either side. With this structure it is also possible to cut the number of multipliers in half when using a reversible generator polynomial [23].

Now consider modifying the systolic encoder to implement Euclid's algorithm, concentrating only on the remainder polynomials $\omega(x)$ and $r(x)$ at this point. First of all, there is no guarantee that the remainder polynomials in the gcd procedure have a leading coefficient of 1 at each iteration, as was implicitly assumed of $g(x)$ for the encoder. Because these remainders will eventually find their way into the $g(x)$ registers, we must compensate for the leading term. Observe that

$$x^r \equiv g_r^{-1} \sum_{i=0}^{r-1} g_i x^i \pmod{g(x)},$$

so the effective coefficients are g_i/g_r. Instead of computing and storing these elements, a simpler approach is to scale the feedback term by g_r^{-1} before sending it up along the registers on the right-hand side. Thus, we will add a g_r register which feeds into a bit-serial inversion unit and into a multiplier in the feedback path. Initially, the syndrome polynomial $\omega(x) = S(x)$ is entered into the multiplier registers, setting $g_i = S_{L+i-1}$ for $i = 1, 2, \ldots, r$ and $g_0 = 0$, since $\deg(S(x)) = r - 1$. If $S_{L+r-1} = 0$, the syndromes must be shifted down until a nonzero term is found; each such shift decreases the degree of ω stored in the controller. Then the coefficients of

$r(x) = x^{2t}$ are fed into the data input of the array. The feedback should be enabled during the first $(\deg(r) - \deg(\omega) + 1)$ cycles, insuring that the output will have degree less than $\deg(\omega)$. For the remaining coefficients, the feedback is turned off, producing the remainder $r(x) \pmod{\omega(x)}$. The first iteration of Euclid's algorithm is now completed.

A major problem arises at this point. How do we move the remainder into the $g(x)$ register and extract $\omega(x)$ from this register to serve as the new $r(x)$? The remainder coefficients emerge in the incorrect order to be fed back up in a staging register for $g(x)$, and since we want to maintain the systolic nature of the structure, it is not possible to transmit values to the cells in the opposite order. Perhaps, by wrapping the array back on itself, these coefficients can effectively be shifted down in the array, but such an approach would require r cycles on every iteration. Also, because roughly $r-1$ cycles will be required to complete the remainder calculation, it seems that the next iteration cannot begin before that time. But if this is the case, our structure will require order r^2 cycles to solve the key equation, which is no better than a sequential processor.

Fortunately, there seems to be a fairly elegant solution to this dilemma. Notice that, when the degree of the remainder drops below that of ω and the feedback is turned off, on the next cycle a zero is present in the delay element which feeds the g_{r-1} multiplier. In other words, the last stage of the feedback shift register on the left has now become a simple shift register, since zero is the input to the adder from the multiplier. On subsequent (interleaved) cycles, inputs to the remaining multipliers also become zero. Thus, after j cycles, the value which emerges from the bottom left register has been unaltered over the last j stages. As a consequence, the stage which is j levels from the bottom is essentially no longer being used at this point, so with some manipulation the next iteration of Euclid's algorithm can begin immediately. Consider the architecture shown in Figure 6-21, which implements all of the changes discussed above. Registers L (for left), R (right) and g correspond directly to the encoder structure, but the accompanying Q and s registers are new. Q holds the coefficients of the current value of $r(x)$, and s is used for control purposes. Each register L, Q and g can accept its new input from two sources, located here on the top and on the left, and the selection will be determined by the contents of the local binary register s_i. When $s_i = 1$, the left input is selected; otherwise, the top input becomes the next value. For encoding purposes, s_i would always be set to zero.

There are several paths here which are not present in the encoder. When the time for a swap arrives, coefficients of the old $\omega(x)$ are moved from the g register to the Q register to become the new $r(x)$, the L register is cleared to begin the next iteration, and all $2t$ coefficients of the new remainder are computed as the sum of the old $r(x)$ and the L register. However, this swapping cannot all occur at once, since the values are not all computed at once. Instead, s_r is set to one, propagating the swap operation upward through the array in the s shift register, one level at a time. The feedback switch never needs to be opened here, since the new iteration begins immediately at the bottom level of the array while the old iteration is completed in the upper section. On each cycle, one additional level of the array, no longer needed for the old computation, becomes involved in the new iteration as a swap occurs. With all of this activity, it is important to remember that everything is still being interleaved to depth two, so alternate problems are considered on alternate clock cycles. On each interleaved cycle, the degree of the largest polynomial is decreased by at least one, so only $2t$ cycles are required to solve the key equation.

On first glance, it seems odd that the Q register can be used to hold the remainders $r(x)$ from both of the (interleaved) gcd algorithms, since there are only $r = 2t$ registers to hold $4t$ coefficients. However, because the next iteration begins immediately, by the time the swap

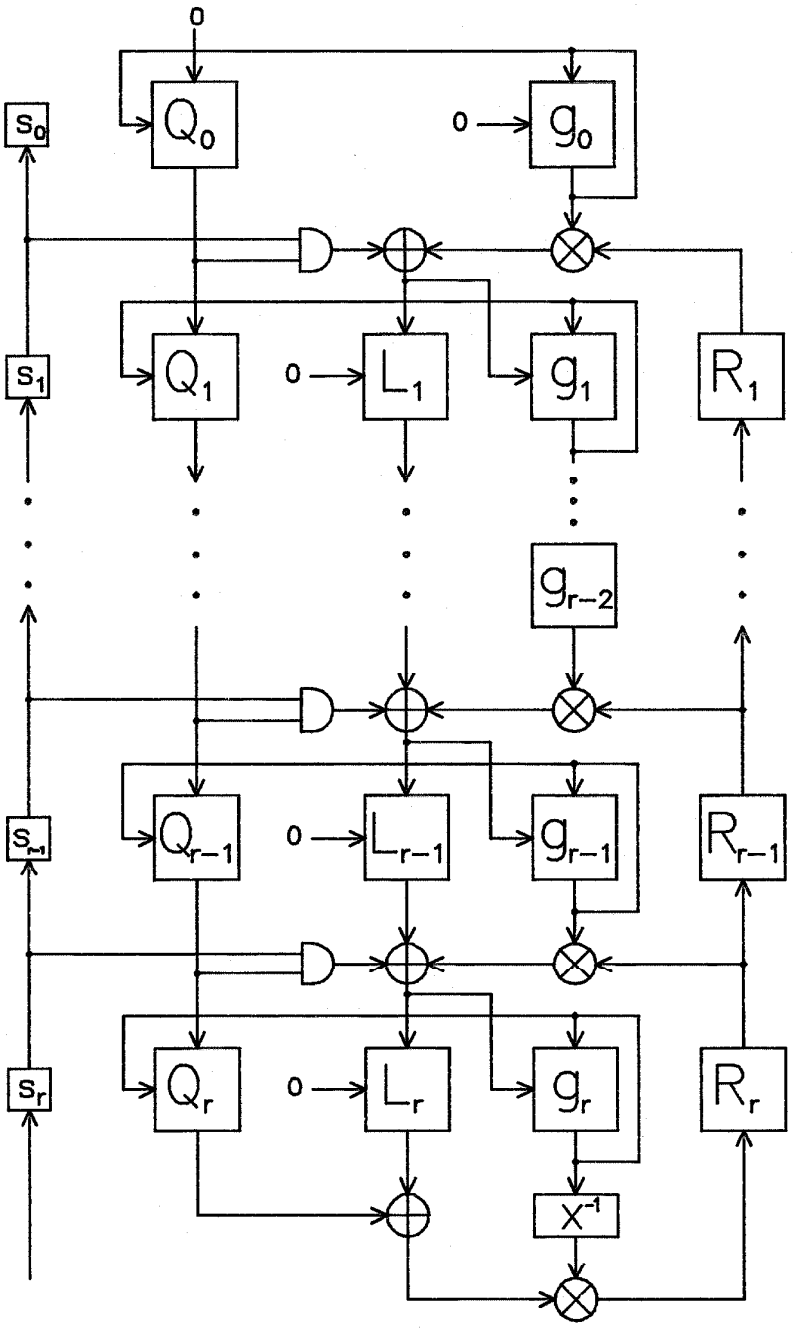

Figure 6-21. Systolic Decoder Structure

signal has propagated up j levels, exactly j values have been emptied from Q into R through the feedback path; i.e., the R registers are effectively extensions of the Q registers, thus providing $4t$ storage cells. When the key equation is solved, the final value of $\omega(x)$ ends up in the g register, where it can be held until the Chien search unit is ready. The new syndromes can then be introduced serially from the top, shifting down in the Q register and being swapped into the g register, allowing the error locator to exit to the Chien search unit. Thus, it is possible to perform syndrome modification for shortened codes in a sequential fashion at the top of the array.

The exact data flow details of this architecture still need to be resolved. In particular, it is important to understand how the auxiliary locator polynomials $\sigma(x)$ and $v(x)$ can be computed using this array. Because the degree of the remainders shrinks throughout the problem, it seems likely that the locator polynomials can be computed in the space vacated by the remainders with a little control overhead, as discussed previously. However, to fully utilize the erasure correction ability of the code, it seems necessary to duplicate the array for the locator polynomials. It is clear that this structure can be used as both an encoder and a Euclidean key equation solver, combining the most attractive features of all the previous architectures. For example, since there is a single central controller, the decoder is free to use reciprocals to cut down on the number of multiplications in each cell, and the degree-tracking state machine needs to be built only once. Because the array operates systolically, no control signals (other than clocks) need be distributed over the entire chip, and it is simple to extend the decoder over several chips, replacing the delay element by the off-chip driver time. Also, an encoder is available on the chip at no extra cost. Unfortunately, interleaving to depth two is required to utilize the key equation unit most efficiently. Again, however, because the other blocks do not need interleaving, the decoder could run without interleaving for codes of high enough rate.

Each of the architectures presented in this section has area $A = O(mt)$, or $A = O(dmt)$ if interleaving to degree d is considered. Again, the clock period has order $P = O(1)$, given the assumptions discussed previously. Although the asymptotic orders for area and time are identical, there are certainly differences in the associated constant factors which will have to be taken into account in deciding which architecture fits best in a given technology. As compared with the Berlekamp-Massey decoder, it seems that the Euclidean algorithm can be implemented with little or no minimum interleaving requirement, and the systolic structures offer great flexibility with respect to using multiple chips in order to handle codes of higher redundancy.

6.8 Berlekamp-Welch Decoder

Both of the algorithms considered thus far involve the power-sum syndromes, but in this section we shall see that the basic structures can be extended to remainder decoding as well. Since the Berlekamp-Welch and the Liu algorithms are so similar, only the former will be discussed here. Not surprisingly, the area, latency, and clock period will be of the same order as those of the previous architectures. In both the reencoder and the Chien search units, it became apparent that remainder decoding methods were only efficient for fixed, non-shortened codes. Examining the modified Berlekamp-Welch algorithm of Figure 5-11, we see that the constants p_k are dependent on the code, so the same restriction seems to hold here.

Because this algorithm cannot be split, storage for four polynomial coefficients must be present in each cell of the key equation solver array, half of which is shown in Figure 6-22. The other half is virtually identical but holds the coefficients of $N(x)$ and $M(x)$. If desired, an inversionless version of the algorithm could also be used to halve the number of multiplies per cell at the cost of an additional multiplexer input to handle the zero discrepancy case. The

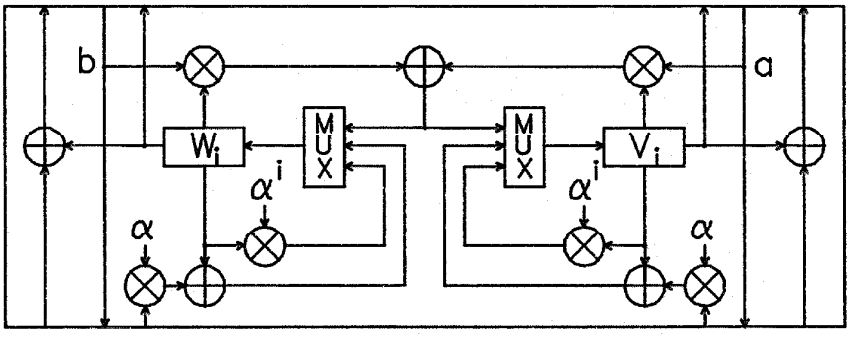

Figure 6-22. Half of Berlekamp-Welch Key Equation Cell

controller sends appropriate select signals to the multiplexers, based on the current values of the parameters D and k, as well as the discrepancies a_k and b_k. The unmodified algorithm requires evaluation of the polynomials at various field locations. A simpler approach in hardware is to apply the operator $\mathcal{D}$, so that all evaluations can be performed at $x = 1$, which is fairly simple to arrange. Figure 5-11 presents this modified form of the algorithm. In the evaluate phase, while the polynomial coefficients are being summed over the array, multiplication by α^i is performed to effect the transformation $\mathcal{D}$. During the update phase, linear combinations of the form $bW_i - aV_i$ are computed by the multipliers. Transformations of the form $(\alpha x - 1)W$ can be performed without a field multiplication, particularly if a dual basis is used, since the bits of αW_{i-1} are available from LFSR register in the multiplier holding W_{i-1}. One awkward feature of this approach is that, at the completion of the algorithm, the polynomials are actually functions of $\alpha^{2t}x$, so the Chien search must start at α^{2t}. Since no information about the check errors is available, the search should then continue with α^{2t+1}, etc., but the error values then emerge in the opposite order from that in which the components were received. A possible fix to this predicament is to remember that the remainder coefficients can be used in any order in the key equation algorithm. In particular, if they are processed in the reverse order and the polynomials are initialized to $W^{(0)}(\alpha^{2t-1}x)$, for example, then the operator $\mathcal{D}' : p(x) \mapsto p(\alpha^{-1}x)$ can be used. At the end of the key equation solution, the polynomials will be functions of $\alpha^{-1}x$, so the Chien search can proceed in the proper sequence.

Clearly this structure has no outstanding advantages over the previous ones considered. In fact, there are several drawbacks, in addition to the limitation of fixed codes. For example, the necessity of evaluating all four polynomials in each iteration would involve considerable global sum hardware. Also, because the algorithm does not split, a full complement of storage and multipliers must be present in each cell. Nonetheless, this architecture is fairly reminiscent of the power syndrome key equation solvers, leading us to believe that any parallel decoder will have a similar structure.

6.9 Results and Applications

For a fixed depth of interleaving, each of the above architectures has $A = O(mt)$ and an overall latency $T = O(m(n + at))$ bit clocks, for some integer a. The model here is one in which components of the received word enter the system sequentially, so the mn term in the latency

Algorithm	Multipliers	Storage	Latency	Min d	Comments
Berlekamp-Massey	2t	6t	6t	3	Split algorithm
Euclid	4t	8t	2t	1	Non-systolic
Euclid (split)	4t	4t	4t	1	Systolic (Kung)
Euclid	8t	8t	2t	1	Systolic (Kung)
Euclid	4t	5t	2t	2	Systolic Encoder
Berlekamp-Welch	4t	8t	4t	3	With inversion

Table 6-1. Area-Time Comparison of Key Equation Architectures

is unavoidable when using bit-serial techniques, and the remaining latency is required to solve the key equation. Recently, El Gamal et al. [25] have shown that, in the case of sequential data input, the area A and latency T are asymptotically related to the redundancy $2t$ and the block length n by

$$A \geq c_1 t \qquad \text{and} \qquad T \geq c_2 n,$$

for some fixed constants c_1 and c_2. In other words, our architectures are within a factor m of the lower bounds in both area and time. The factor of m in the latency is related to bit-serial computation, but notice that parallel arithmetic decreases the latency by this factor while the area increases proportionately. Typically $n = 2^m - 1$, or $m \approx \log n$, so in terms of area-time product the lower bound is exceeded here by roughly $(\log n)^2$. Although these structures may not be optimal, they do provide an existent upper bound.

However, perhaps the most important characteristic of our decoders, as opposed to the sequential architectures examined by Cohen, is that the throughput, measured in decoded bits per second, does not decrease with t or m. That is, a bit-serial decoder with a 16 MHz clock has a throughput of 16 Mbits/sec, regardless of the redundancy or the field size. By contrast, the GF1 decoder over GF(256) running at a 16 MHz clock rate performs field multiplications at the same speed as a bit-serial multiplier running at 128 MHz, but the throughput for a (255,239) code is less than 4 Mbits/sec, and performance degrades for higher redundancies. It is clear that parallelism must be employed to build high performance decoders for Reed-Solomon codes of non-trivial redundancy.

In practice, however, codes of interest involve fairly small m and t. Of much greater concern than the asymptotic results are the constant factors accompanying the order estimates. We have seen that the syndrome/remainder generation and the Chien search units are basically identical in all the architectures. Table 6-1 summarizes the results for various key equation solver structures presented above. Controller area is not taken into account, since it does not scale with the redundancy, except in the case of Kung's gcd array. The second and third columns give a relative area estimate by counting the total number of full multipliers and the total number of coefficients (field elements) which must be stored in the array. The latency is the maximum number of interleaved cycles required to complete the key equation solution, with the minimum depth of interleaving d for the architecture given in the following column.

In terms of latency, the Euclidean algorithms have a distinct advantage because of the simplicity of their discrepancy term. If area is the major concern, the Berlekamp-Massey algorithm seems to have a slight edge, especially because multipliers are considerably larger than storage registers. The systolic approaches have advantages already discussed; in particular, the ease with which the algorithm can be spread over multiple chips somewhat reduces the area

penalty with respect to the Berlekamp-Massey architecture. Our goal here is not find the optimal approach, since many system factors must be taken into account in such a decision. However, the analysis in this chapter should be helpful in selecting an architecture for implementation.

At this point, it seems appropriate to stop and reflect on the possible implications of a single-chip Reed-Solomon encoder/decoder. How could such a chip be used? Many straightforward and important applications are possible in magnetic or optical data storage and in communication systems, but we shall leave these details to the engineers. From a more philosophical point of view, the basic unit of computation in decoding hardware would jump from a multiply or a memory reference to an entire decoder. In fact, if the data rates were on the order of a few megabits/second or below, the chip could have as few as 6 pins: power, ground, clock, word sync, data in, errors out, although such packaging may be neither desirable nor practical. Nonetheless, the hardware and power cost associated with decoding (and encoding) clearly would be reduced by orders of magnitude, while the worst-case throughput increases significantly. One immediate consequence is that codes with fairly high redundancy, say greater than 16, which are now feasible only in cost-insensitive or low-speed applications, could be implemented routinely, utilizing the full power of Reed-Solomon codes.

Perhaps more intriguing, however, is the question of how to utilize many such decoder chips to achieve formerly impossible speed or reliability goals. For example, interleaving can be performed at the chip level, with simple high-speed logic routing sequential inputs to adjacent chips and collating the results. This technique can be used to achieve a linear increase in throughput, at the cost of greater depths of interleaving. Alternatively, entire words, entering the system at high speed, can be buffered and presented to decoder chips on a round-robin basis, again effecting a linear improvement in decoding speed. There is no required depth of interleaving in this case, although the buffering logic would be considerably more complex than in the former case. Also, consider a matrix of field elements in which both the rows and columns form RS codewords. Decoding and encoding of a matrix Reed-Solomon code could be accomplished using a few chips, adding a new dimension (literally) of codes which are feasible to implement.

Another very important application could come in the area of soft decision decoding. For example, Weldon and Chase have proposed algorithms requiring multiple hard decision decoders in an attempt to utilize soft information in block codes [15]. These algorithms seem to perform encouragingly like true maximum-likelihood decoding, but previously it has been feasible to apply such ideas only to fairly simple (e.g., binary) codes. One possible approach would be to identify a set of least reliable symbols in the RS word, using analog information from the demodulator. In general, it would be possible to choose a set larger than the redundancy of the code. By erasing distinct subsets of these components and presenting each problem to a different chip, several estimates of the transmitted word could be produced and graded by comparison with the actual received word. If the number of erased symbols is limited to several less than the redundancy, some errors in addition to the erasures can be corrected and the probability of miscorrection can be made negligible. More research in this area will prove useful when single-chip decoders actually become available.

Chapter 7

Conclusion

Beginning from the elementary concepts of abstract algebra and coding theory, we have finally worked our way up to a thorough understanding of the arithmetic and algorithmic structures necessary to implement Reed-Solomon decoders efficiently on an integrated circuit. Several significant theoretical and practical results have been presented, including the criterion for the existence of self-dual bases, methods for bit-serial inversion, and techniques to allow a single chip to handle easily codes of varying block length and redundancy without affecting performance. For a code of redundancy $2t$ over GF(2^m), our proposed decoders have area $A = O(mt)$ and clock period $P = O(1)$, with the throughput rate in bits per second equal to the clock frequency of the chip. Enough preliminary work has been done to show that it is indeed feasible to construct such a chip, using technology which is presently available. This problem is a excellent example of how a background in abstract mathematics can be merged with a knowledge of VLSI design to produce a high performance system. Without a thorough familiarity with both fields it would have been difficult to arrive at these conclusions.

Although the emphasis here has been on bit-serial computation, many of our results can be applied to the case of parallel arithmetic as well. As technology improves, area will become less of a concern, and fully parallel multiplication, addition, and inversion may become the techniques of choice for high performance decoding. There are also many combinations of bit-serial and parallel computation, such as pipelined parallel structures, which will prove useful. It is hoped that the bit-serial methods derived in this thesis can serve as a guide to making intelligent decisions in these tradeoffs.

Research questions are often stimulated by the advent of new technology or tools. Although Berlekamp conceived the idea of a parallel decoder engine almost two decades ago, not until the VLSI era did it become feasible to consider implementation of such a machine. By the same token, a single chip decoder will undoubtedly lead to many other interesting applications and, ultimately, new questions.

Appendix A

Normal-Basis Multiplication

Let the product $z = xy$ be given in a normal basis $\mathrm{B} = \{\mu^{2^i}\}$, and let x be represented in the B basis and y be represented in some other normal basis $\mathrm{P} = \{\omega^{2^j}\}$. Then,

$$x = \sum_{i=0}^{m-1} x_i \mu^{2^i} \qquad \text{and} \qquad y = \sum_{j=0}^{m-1} y_j \omega^{2^j},$$

and

$$z = xy = \sum_{i,\,j=0}^{m-1} x_i y_j \mu^{2^i} \omega^{2^j}.$$

Let $\mathrm{D} = \{\alpha^{2^i}\}$ be the basis dual to B, so the first bit of the product z is given by

$$z_0 = \mathrm{Tr}(\alpha_0 xy) = \sum_{i,\,j=0}^{m-1} x_i y_j \mathrm{Tr}\Big(\alpha_0 \mu^{2^i} \omega^{2^j}\Big). \tag{A.1}$$

Now consider the case $y = 1$. Clearly, since P is a basis, we must have $Tr(\omega) = 1$, so in this case $y_j = 1$ for $j = 0, 1, \ldots, m-1$. But since $y = 1$, $z = x$, so we must have $z_0 = x_0$, regardless of the value of x. Recasting (A.1) in this light, we find

$$z_0 = x_0 = \sum_{i=0}^{m-1} x_i \sum_{j=0}^{m-1} \mathrm{Tr}\Big(\alpha_0 \mu^{2^i} \omega^{2^j}\Big). \tag{A.2}$$

We break (A.2) down into two cases: for $i = 0$,

$$1 = \sum_{j=0}^{m-1} \mathrm{Tr}\Big(\alpha_0 \mu \omega^{2^j}\Big),$$

or x_0 is involved in an odd number of product terms. But for $i = 1, 2, \ldots, m-1$,

$$0 = \sum_{j=0}^{m-1} \mathrm{Tr}\left(\alpha_0 \mu^{2^i} \omega^{2^j}\right),$$

so x_i is involved in an even number of product terms. Note that each component x_i must be able to have some effect on the first bit of the product, so in fact each x_i is involved in at least two product terms, for $i \neq 0$. Therefore, there are at least $1 + 2(m-1) = 2m - 1$ product terms involved in bit-serial multiplication using normal bases.

Massey [38] has shown, by using the distributive laws, that the number of actual GF(2) multiplications can be reduced to m; however, such transformations do not decrease the number of XOR operations below the minimum of $2m - 2$. Further, for VLSI implementation, the gate count is often not as important as the wireability of the logic. Note that factoring the expression using the distributive law corresponds roughly to performing a basis change, and such a structure tends to have area proportional to m^2. To obtain a regular structure for their normal-basis multiplier, a design team at the Jet Propulsion Laboratory [31] found it necessary to use a PLA structure, which has size m^2. Thus, with respect to a dual-basis multiplier, an area penalty of roughly two will be paid due to gate count, and wiring is likely to increase this cost greatly.

Appendix B

Gilbert Model Probability Computation

Consider a Reed-Solomon code of block length n over GF(2^m), interleaved to depth d. On the bit level, the data appear as in figure B-1. Let us assume that the channel error statistics can be approximated by the Gilbert model, as shown in figure 3-5. The average burst spacing is $1/\lambda$ bits, and the average burst length is $1/\mu$ bits. We then proceed to compute the error probabilities in several steps. Our computation will consider only one of the interleaved codewords, so let us define an *interleaved cycle* to be a sequence of d characters, consisting of a component from the codeword of interest followed by $m(d-1)$ interleaved bits. Let $P_i(k)$ be the probability that, after i interleaved cycles, exactly k character errors have occurred and that the channel is in the good state. Similarly, let $Q_i(k)$ be the probability that, after i interleaved cycles, exactly k character errors have occurred and that the channel is in a burst. Since at most one character error can occur in a single interleaved cycle, the process is clearly a Markov chain, so there is a linear relationship:

$$\begin{aligned} P_{i+1}(k) &= u_1 P_i(k) + u_2 Q_i(k) + u_3 P_i(k-1) + u_4 Q_i(k-1) \\ Q_{i+1}(k) &= v_1 P_i(k) + v_2 Q_i(k) + v_3 P_i(k-1) + v_4 Q_i(k-1)\,, \end{aligned} \tag{B.1}$$

where the u_i and v_i are the Markov state transition probabilities. On entry to a block, $P_0(k) = Q_0(k) = 0$ for all $k \neq 0$; estimates for the values $P_0(0)$ and $Q_0(0)$ will be derived below. Given the transition probabilities, repeated application of (B.1) can be used to compute the block error probabilities $P_n(k)$ and $Q_n(k)$. For a t error correcting code, the probability of decoder error is given by

$$P_E = \sum_{k=t+1}^{n} (P_n(k) + Q_n(k)).$$

There are two independent phases of an interleaved cycle. The first stage consists of the m bits in the codeword of interest, where bit errors can affect the total error count. Let us see how the initial probabilities $P_i(k)$ and $Q_i(k)$ are split as the channel proceeds through the character. Obviously, the transitions between adjacent k are independent of i. Let us define

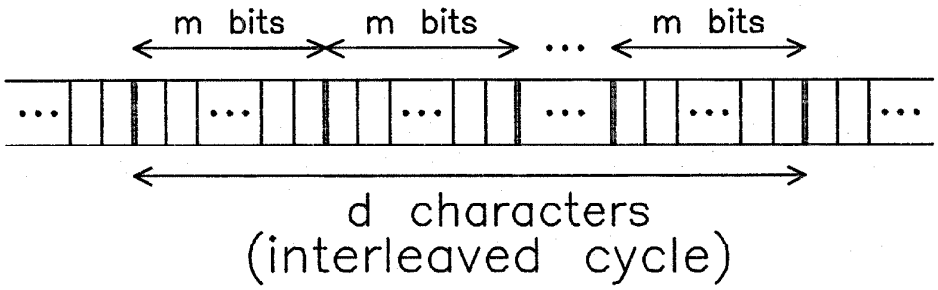

Figure B-1. RS Code Interleaved to Depth d

p_j as the probability of being in the good state after j bits of the current codeword character, with no bit errors having occurred. Similarly, define q_j as the probability of being in the bad state after j bits of the current codeword character, with no bit errors having occurred. The values p_0 and q_0 will be related to the $P_i(k)$ and $Q_i(k)$ later. Let $\tilde{p}_j$ and $\tilde{q}_j$ be the corresponding probabilities with at least one bit error having occurred; clearly $\tilde{p}_0 = \tilde{q}_0 = 0$. The transition equations obtained from the Gilbert model are then

$$\begin{aligned} p_{i+1} &= (1-\lambda)p_i + \frac{1}{2}\mu q_i \\ q_{i+1} &= \lambda p_i + \frac{1}{2}(1-\mu)q_i \\ \tilde{p}_{i+1} &= \frac{1}{2}\mu q_i + (1-\lambda)\tilde{p}_i + \mu\tilde{q}_i \\ \tilde{q}_{i+1} &= \frac{1}{2}(1-\mu)q_i + \lambda\tilde{p}_i + (1-\mu)\tilde{q}_i . \end{aligned} \tag{B.2}$$

Equation (B.2) can also be expressed as a matrix equation, $\mathbf{p}_j = H^j\mathbf{p}_0$, where

$$H = \begin{pmatrix} (1-\lambda) & \frac{1}{2}\mu & 0 & 0 \\ \lambda & \frac{1}{2}(1-\mu) & 0 & 0 \\ 0 & \frac{1}{2}\mu & (1-\lambda) & \mu \\ 0 & \frac{1}{2}(1-\mu) & \lambda & (1-\mu) \end{pmatrix}.$$

The matrix H has four distinct real eigenvalues, so H^j could be computed in closed form. Unfortunately, two of the eigenvalues involve radicals, and the expressions tend to be rather messy. For practical purposes, the matrix exponentation can be performed numerically. Given $A_{ij} = (H^m)_{ij}$, the result is

$$\begin{aligned} p_m &= A_{11}p_0 + A_{12}q_0 \\ q_m &= A_{21}p_0 + A_{22}q_0 \\ \tilde{p}_m &= A_{31}p_0 + A_{32}q_0 \\ \tilde{q}_m &= A_{41}p_0 + A_{42}q_0 . \end{aligned} \tag{B.3}$$

If we define $R_i(k)$ and $S_i(k)$ to be the probabilities corresponding to $P_i(k)$ and $Q_i(k)$, respectively, after i interleaved cycles and one additional codeword character, then (B.3) implies

$$\begin{aligned} R_i(k) &= r_1 P_i(k) + r_2 Q_i(k) + r_3 P_i(k-1) + r_4 Q_i(k-1) \\ S_i(k) &= s_1 P_i(k) + s_2 Q_i(k) + s_3 P_i(k-1) + s_4 Q_i(k-1), \end{aligned} \tag{B.4}$$

where

$$\begin{aligned} r_1 &= A_{11}, & r_2 &= A_{12}, & r_3 &= A_{31}, & r_4 &= A_{32}, \\ s_1 &= A_{21}, & s_2 &= A_{22}, & s_3 &= A_{41}, & s_4 &= A_{42}. \end{aligned}$$

The second independent stage occurs when the character from the codeword of interest is completely received, and the remaining $m(d-1)$ interleaved bits are being processed by the channel. Since no additional codeword errors can occur here, there can be no transitions between probabilities involving distinct values of k. It can readily be seen that the bit transition equations are given by the lower 2×2 matrix of H, involving only $\tilde{p}$ and $\tilde{q}$. In other words,

$$\begin{aligned} p_{i+1} &= (1-\lambda)p_i + \mu q_i \\ q_{i+1} &= \lambda p_i + (1-\mu)q_i. \end{aligned} \tag{B.5}$$

Fortunately, this set of equations can be solved explicitly, because (B.5) has eigenvalues 1 and $1-\mu-\lambda$. After some algebra, it can be shown that

$$\begin{aligned} p_j &= \delta^{-1}\big(\mu(p_0+q_0) + (1-\delta)^j(\lambda p_0 - \mu q_0)\big) \\ q_j &= \delta^{-1}\big(\lambda(p_0+q_0) - (1-\delta)^j(\lambda p_0 - \mu q_0)\big), \end{aligned} \tag{B.6}$$

where $\delta = \mu + \lambda$. Observe that, as j gets large, the second term drops out, since $|1-\delta| < 1$. Thus, the probability that any bit in a random stream is part of an error burst, assuming that $p_0 + q_0 = 1$, is given by $\mu/(\lambda+\mu)$; we may therefore assume that

$$P_0(0) = 1 - Q_0(0) = \frac{\mu}{\lambda+\mu}.$$

From (B.6), it follows that

$$\begin{aligned} P_{i+1}(k) &= \delta^{-1}\big(\mu(R_i(k)+S_i(k)) + (1-\delta)^j(\lambda R_i(k) - \mu S_i(k))\big) \\ Q_{i+1}(k) &= \delta^{-1}\big(\lambda(R_i(k)+S_i(k)) - (1-\delta)^j(\lambda R_i(k) - \mu S_i(k))\big). \end{aligned} \tag{B.7}$$

Let us define $\beta = (1-\delta)^{m(d-1)}$. Combining the results (B.4) and (B.7), we then compute the transition probabilities over an entire interleaved cycle. For $j = 1, 2, 3, 4$,

$$\begin{aligned} u_j &= \delta^{-1}\big((\mu+\lambda\beta)r_j + \mu(1-\beta)s_j\big) \\ v_j &= \delta^{-1}\big(\lambda(1-\beta)r_j + (\lambda+\mu\beta)s_j\big). \end{aligned}$$

These coefficients can all be calculated numerically. So, given the initial conditions $P_0(0)$ and $Q_0(0)$ as derived above, with $P_0(k) = Q_0(k) = 0$ for $k \neq 0$, repeated applications of the transition equations

$$\begin{aligned} P_{i+1}(k) &= u_1 P_i(k) + u_2 Q_i(k) + u_3 P_i(k-1) + u_4 Q_i(k-1) \\ Q_{i+1}(k) &= v_1 P_i(k) + v_2 Q_i(k) + v_3 P_i(k-1) + v_4 Q_i(k-1) \end{aligned} \tag{B.8}$$

can be used to compute the final output probabilities $P_n(k)$ and $Q_n(k)$. If we define $P_i(-1) = Q_i(-1) = 0$, then (B.8) is valid for all $k \geq 0$. Then, assuming the decoder can correct t character errors, the block error probability is given by

$$P_E = \sum_{k=t+1}^{n} (P_n(k) + Q_n(k)).$$

The output bit error probability can be approximated by noting that, if $k > 2t$ character errors occur, then on the average half of the bits in these characters are in error. In other words,

$$p_e = \sum_{k=t+1}^{n} \frac{k}{2n} (P_n(k) + Q_n(k)).$$

This is the method employed to compute the curves of figure 3-6.

Often, error probabilities are computed for interleaving to infinite depth. In this case, the computation can be simplified to a binomial distribution, since each character has an independent probability of being in error. From the above arguments, it is clear that such a computation involves setting $\beta = 0$ in the expression for the transition coefficients. Finite interleaving approaches the infinite ideal as $\beta = (1 - \lambda - \mu)^{m(d-1)}$ becomes small with respect to the output block error probability. This fact could be used to select an appropriate interleaving depth if almost independent character errors are desired.

Several plots are presented in the following pages to show how increasing the depth of interleaving improves the error protection of RS codes. Obviously, as the depth of interleaving d increases, the error probability for a given λ and μ decreases; the net effect is roughly a scaling of the burst length axis. The data are presented for a particular code, the (255,239) RS code over GF(256), in two distinct formats. In figures B-2 through B-8, contours of a given output bit error probability are displayed for various interleaving depths. Then, using the same data, contours of constant output bit error probability are plotted for a given depth of interleaving in figures B-9 through B-14. Again, these data should be interpreted qualitatively, since no actual channel is modelled perfectly by the Gilbert model. However, these curves provide some understanding of the power of RS codes and of interleaving.

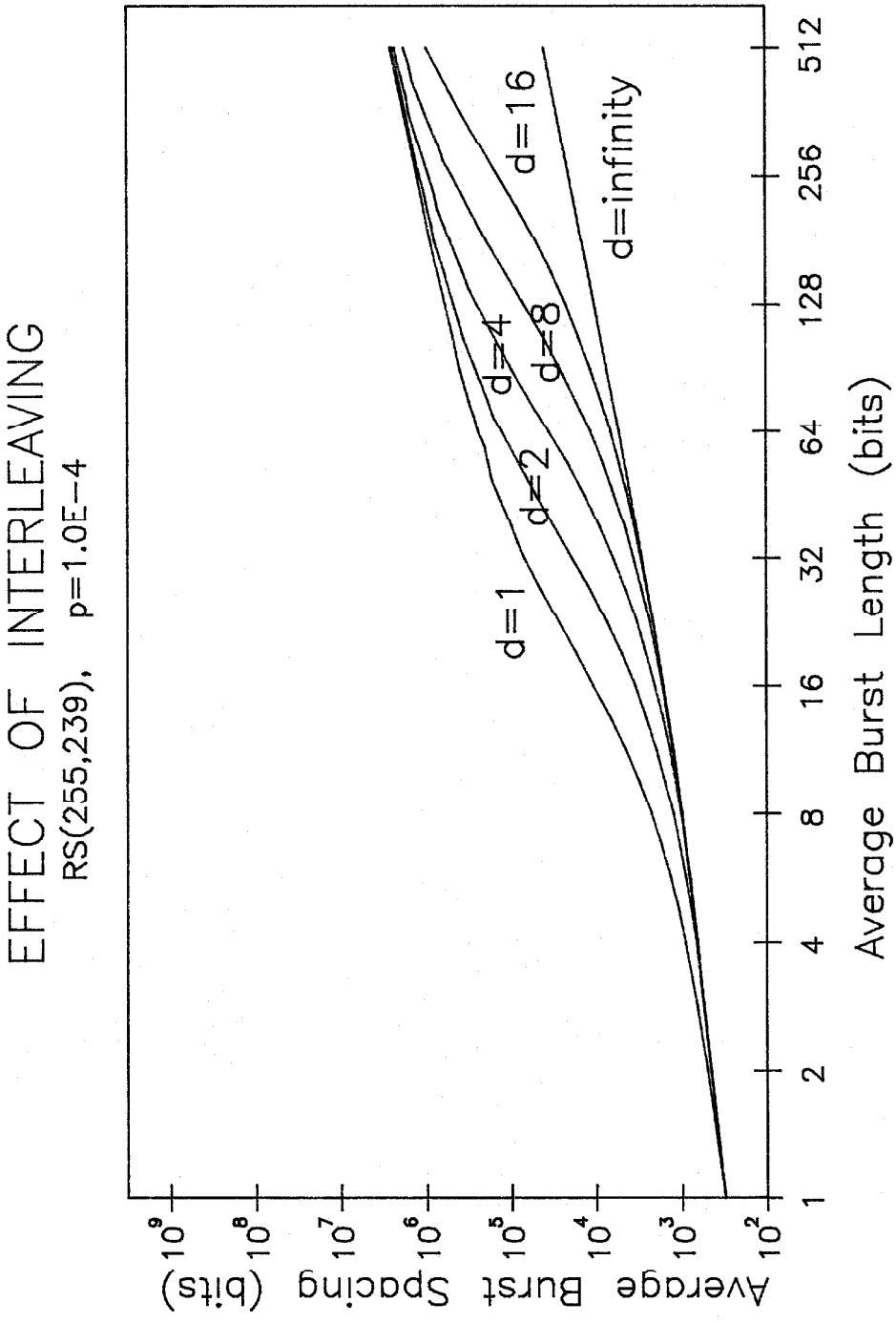

Figure B-2. Effect of Interleaving, $p_e = 10^{-4}$

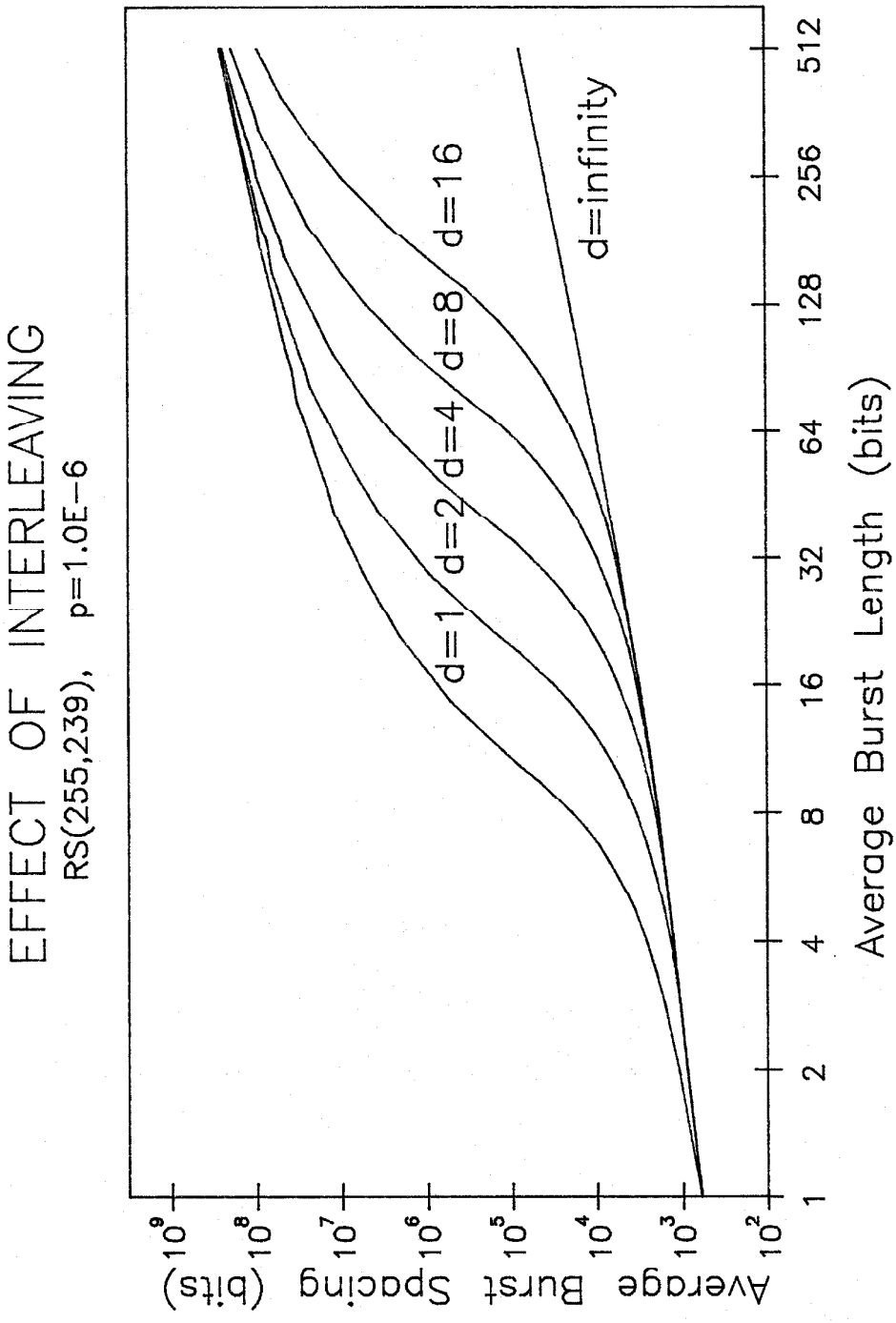

Figure B-3. Effect of Interleaving, $p_e = 10^{-6}$

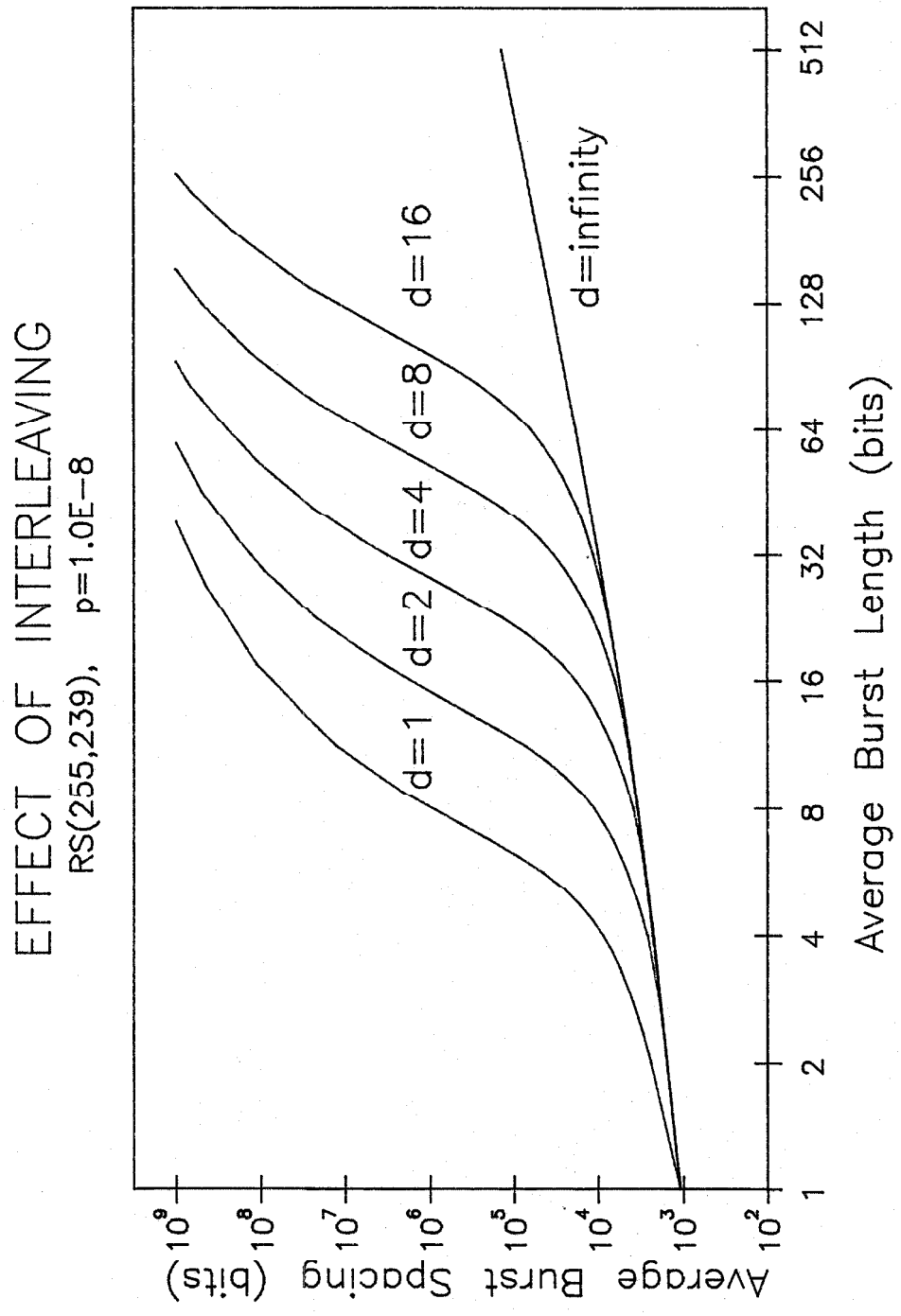

Figure B-4. Effect of Interleaving, $p_e = 10^{-8}$

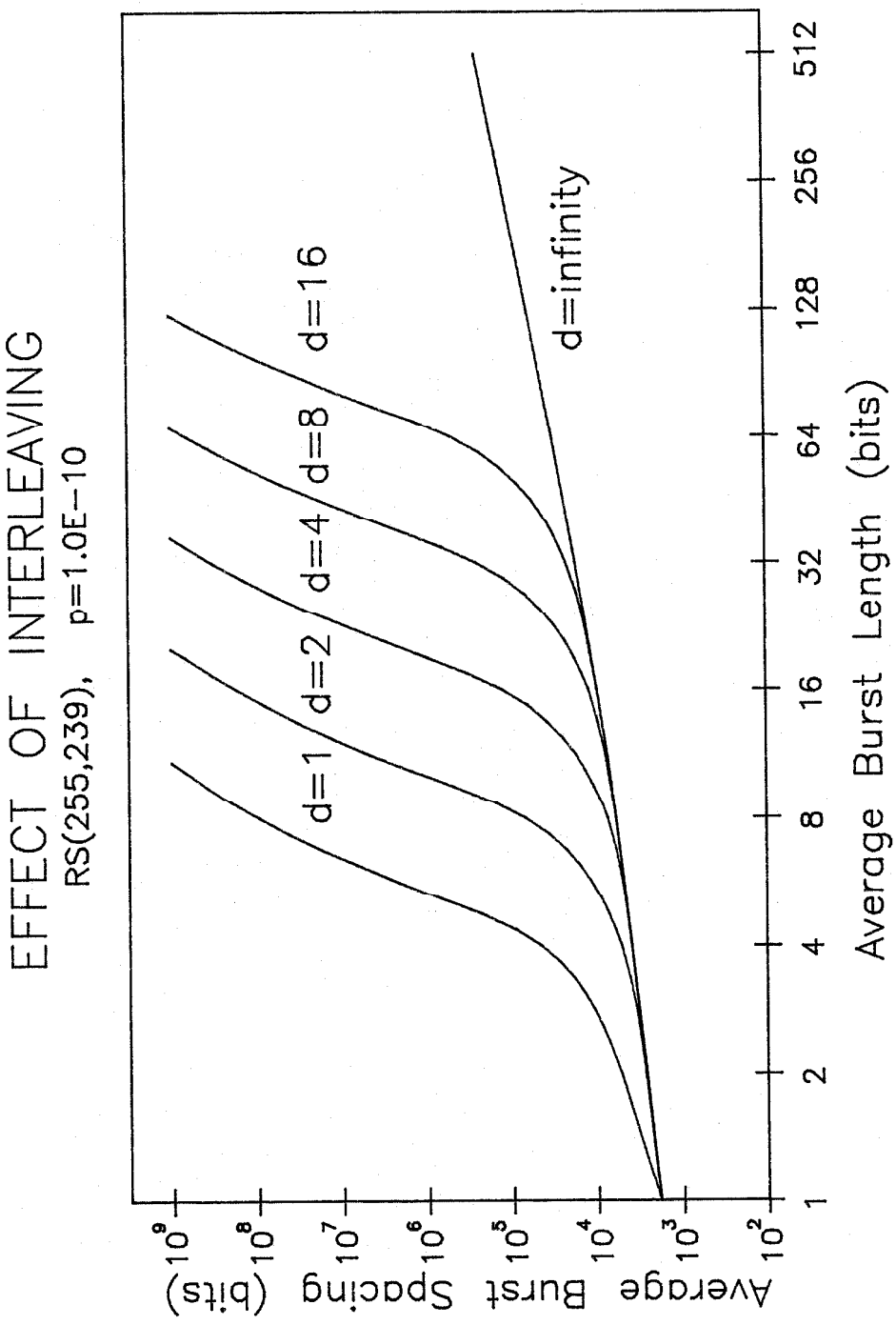

Figure B-5. Effect of Interleaving, $p_e = 10^{-10}$

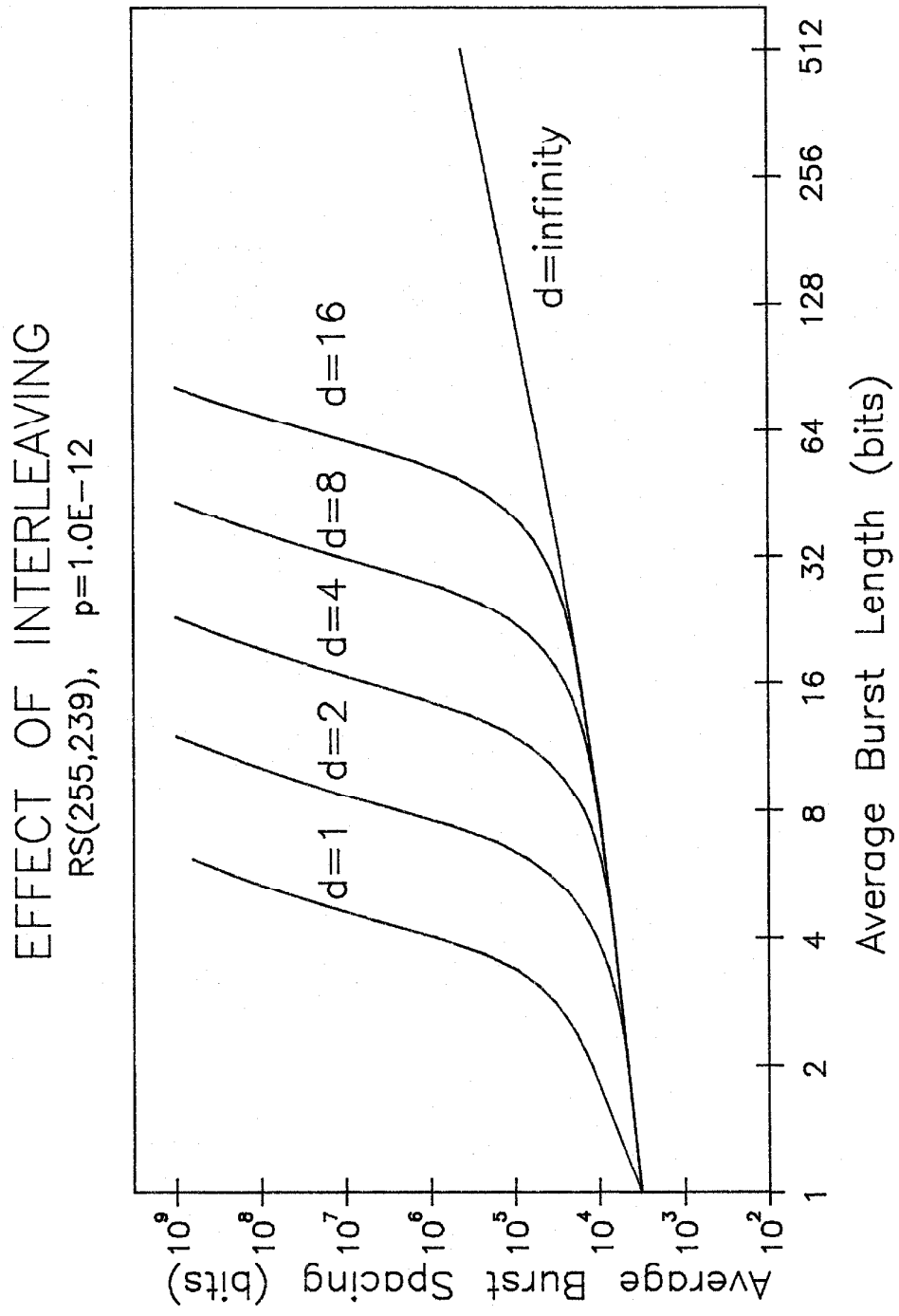

Figure B-6. Effect of Interleaving, $p_e = 10^{-12}$

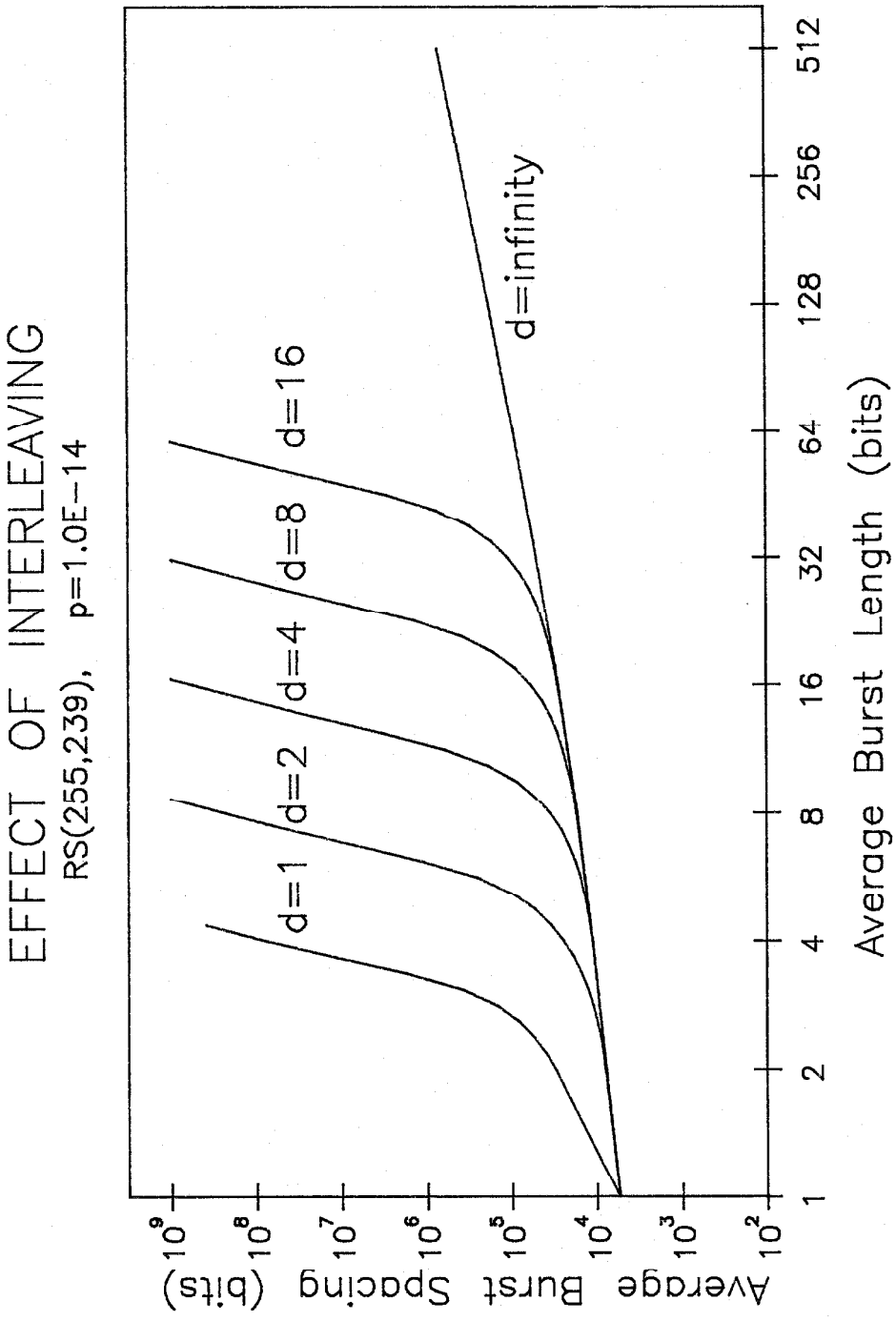

Figure B-7. Effect of Interleaving, $p_e = 10^{-14}$

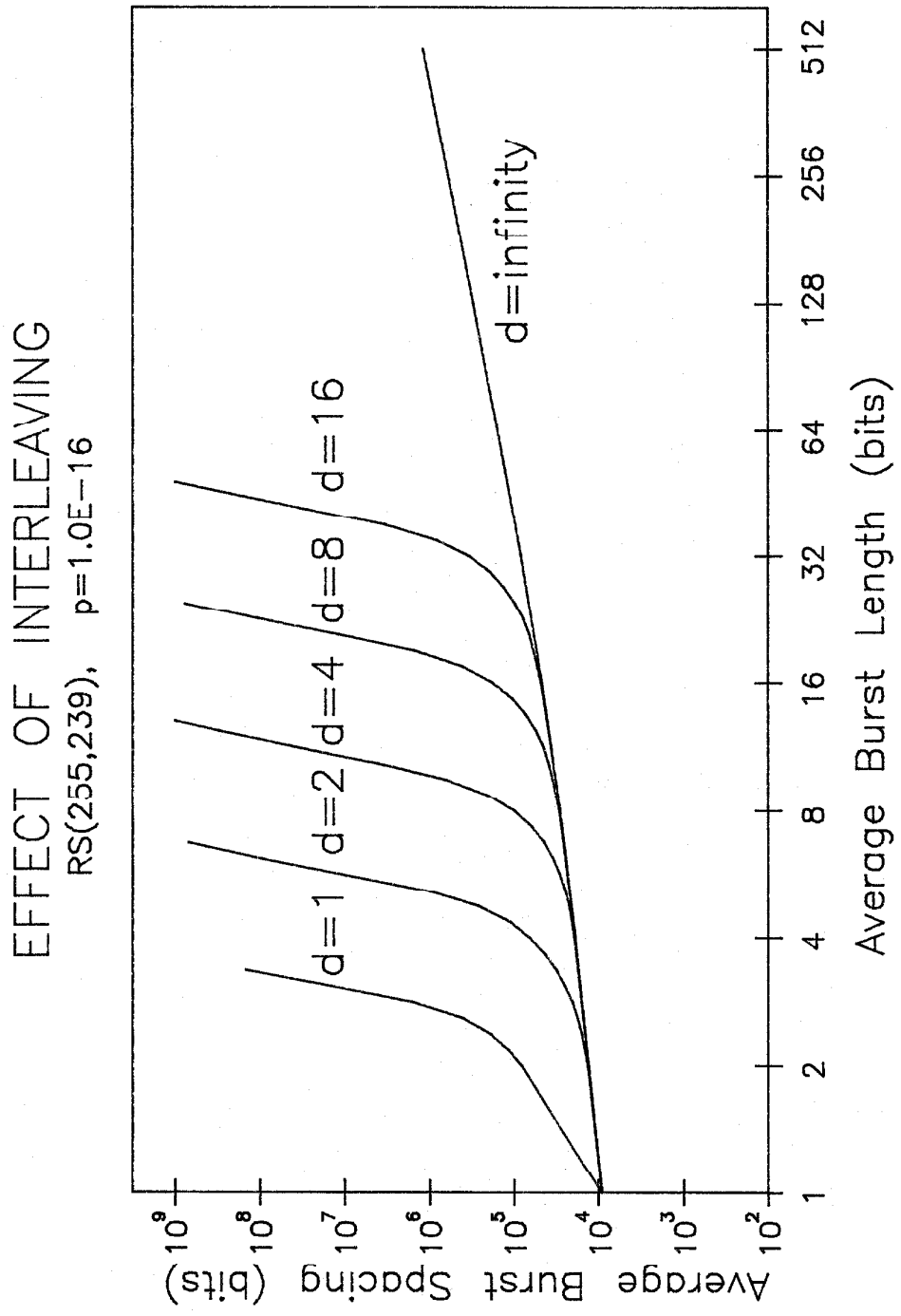

Figure B-8. Effect of Interleaving, $p_e = 10^{-16}$

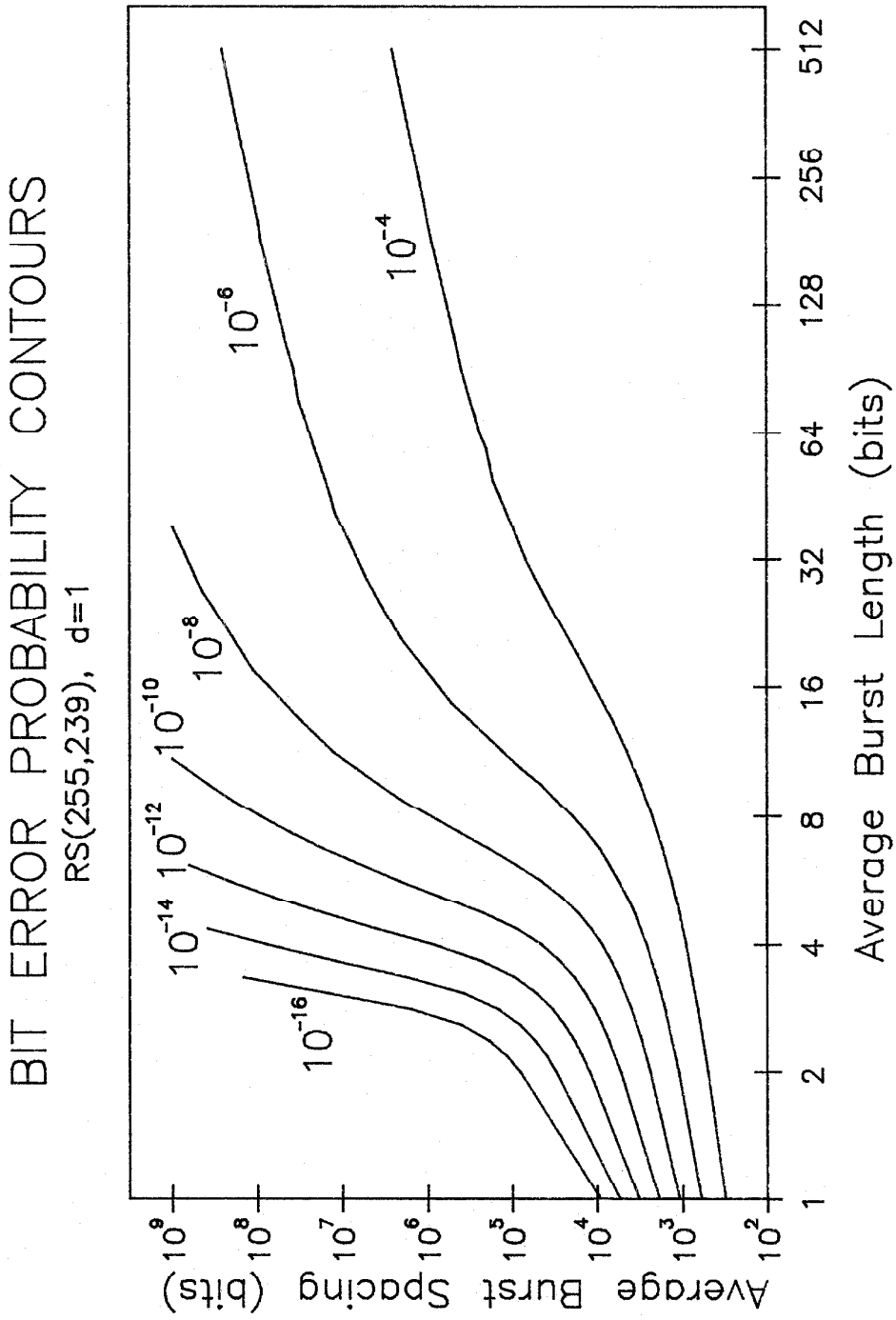

Figure B-9. Bit Error Probability Contours, $d = 1$

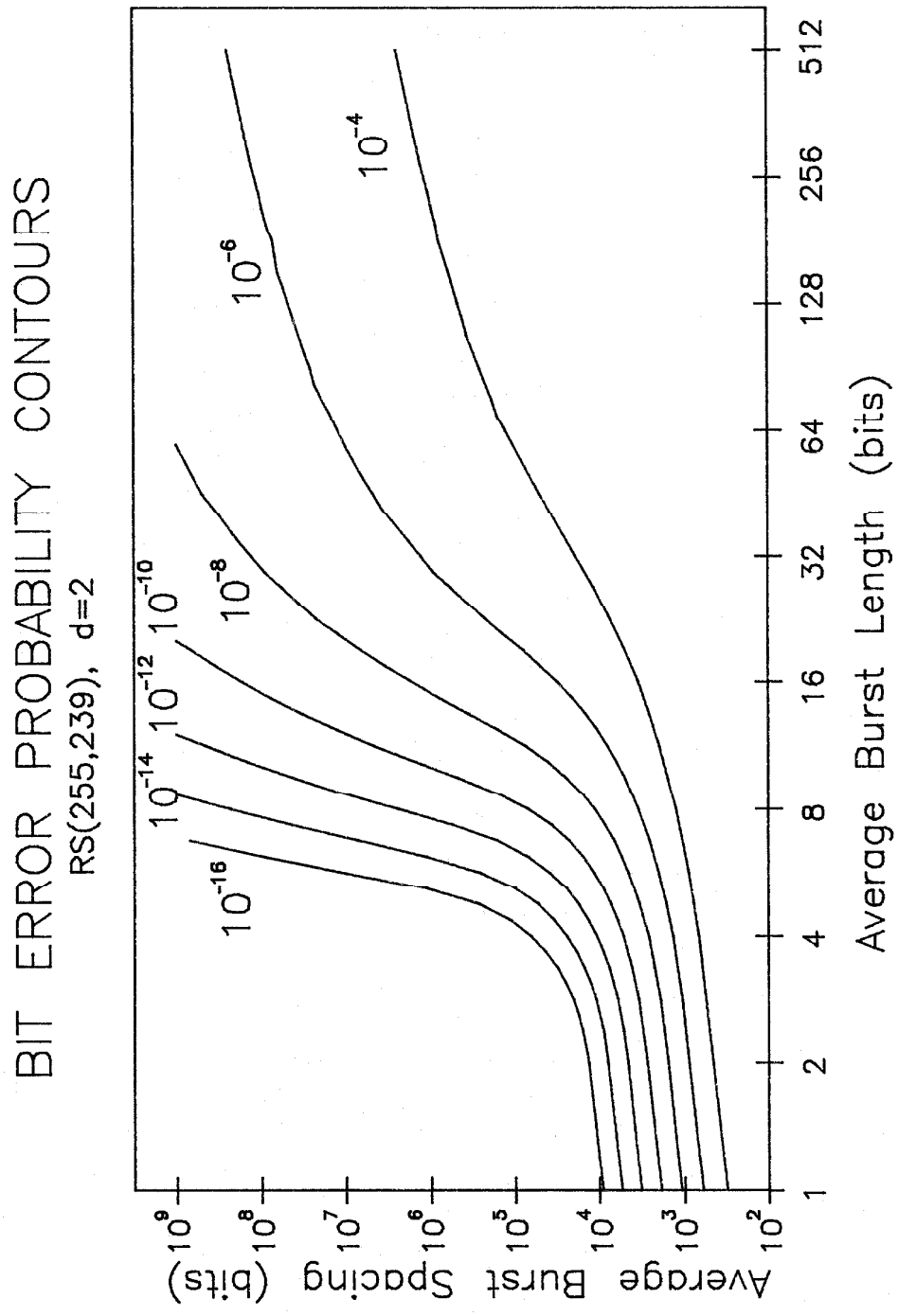

Figure B-10. Bit Error Probability Contours, $d = 2$

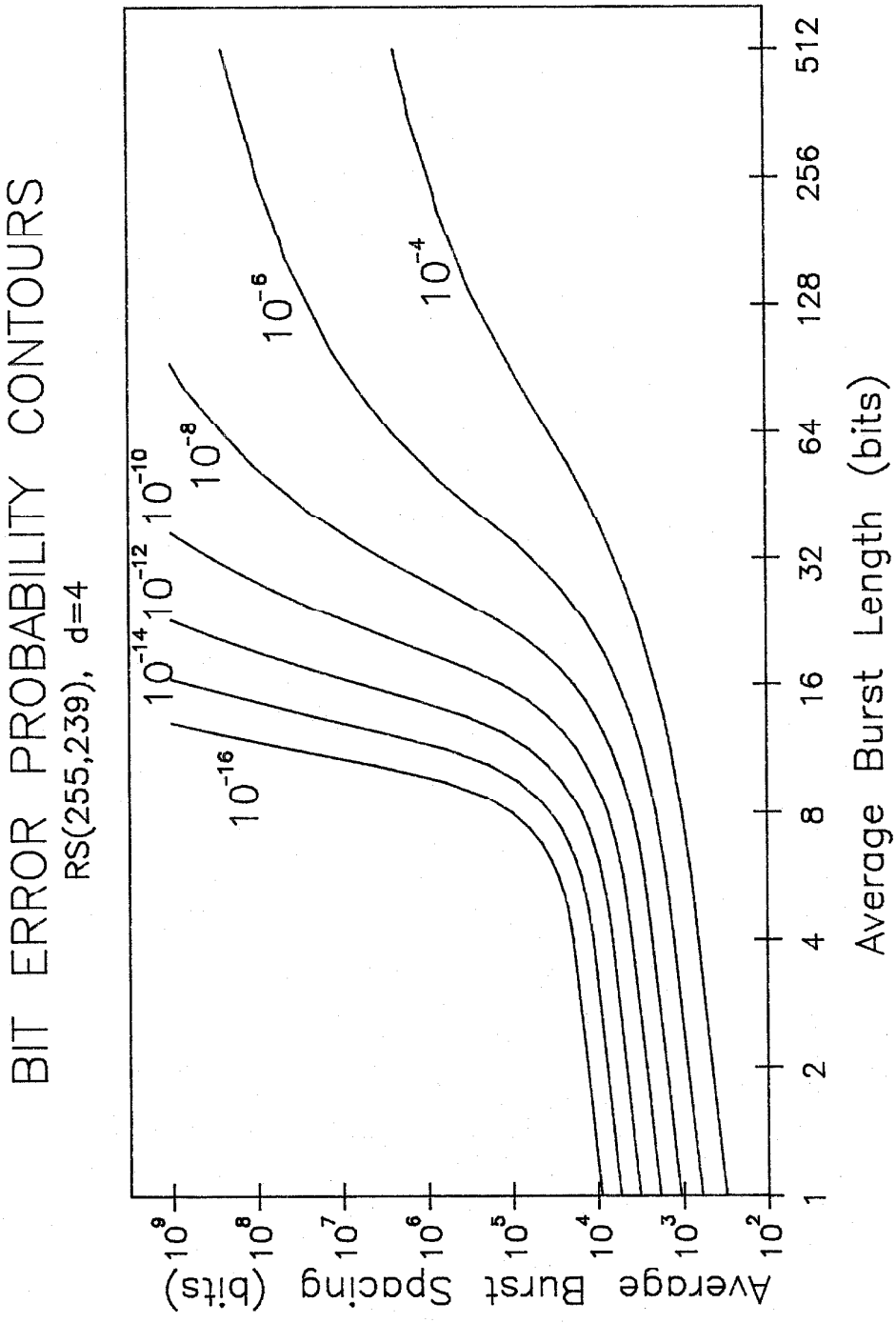

Figure B-11. Bit Error Probability Contours, $d = 4$

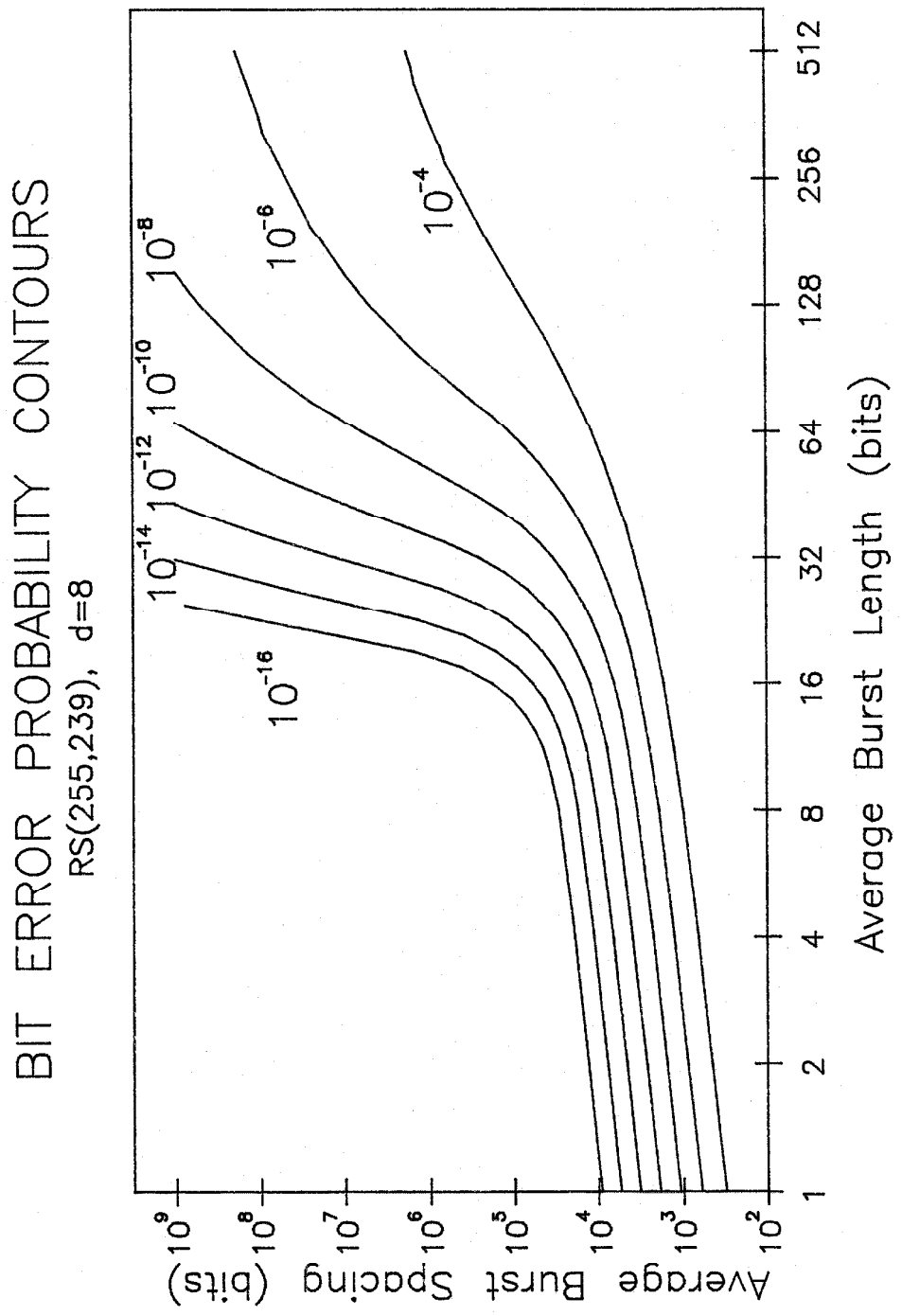

Figure B-12. Bit Error Probability Contours, $d = 8$

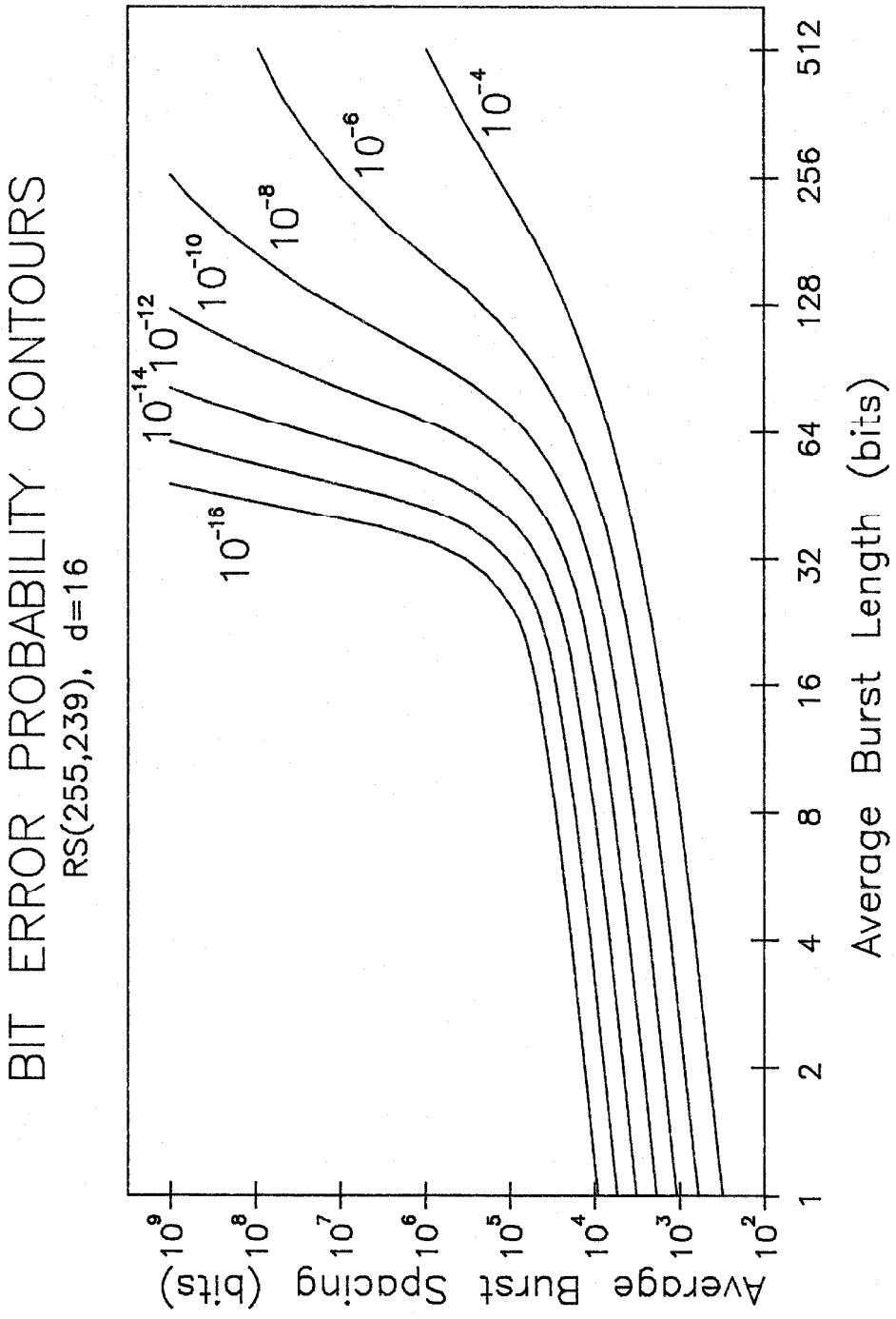

Figure B-13. Bit Error Probability Contours, $d = 16$

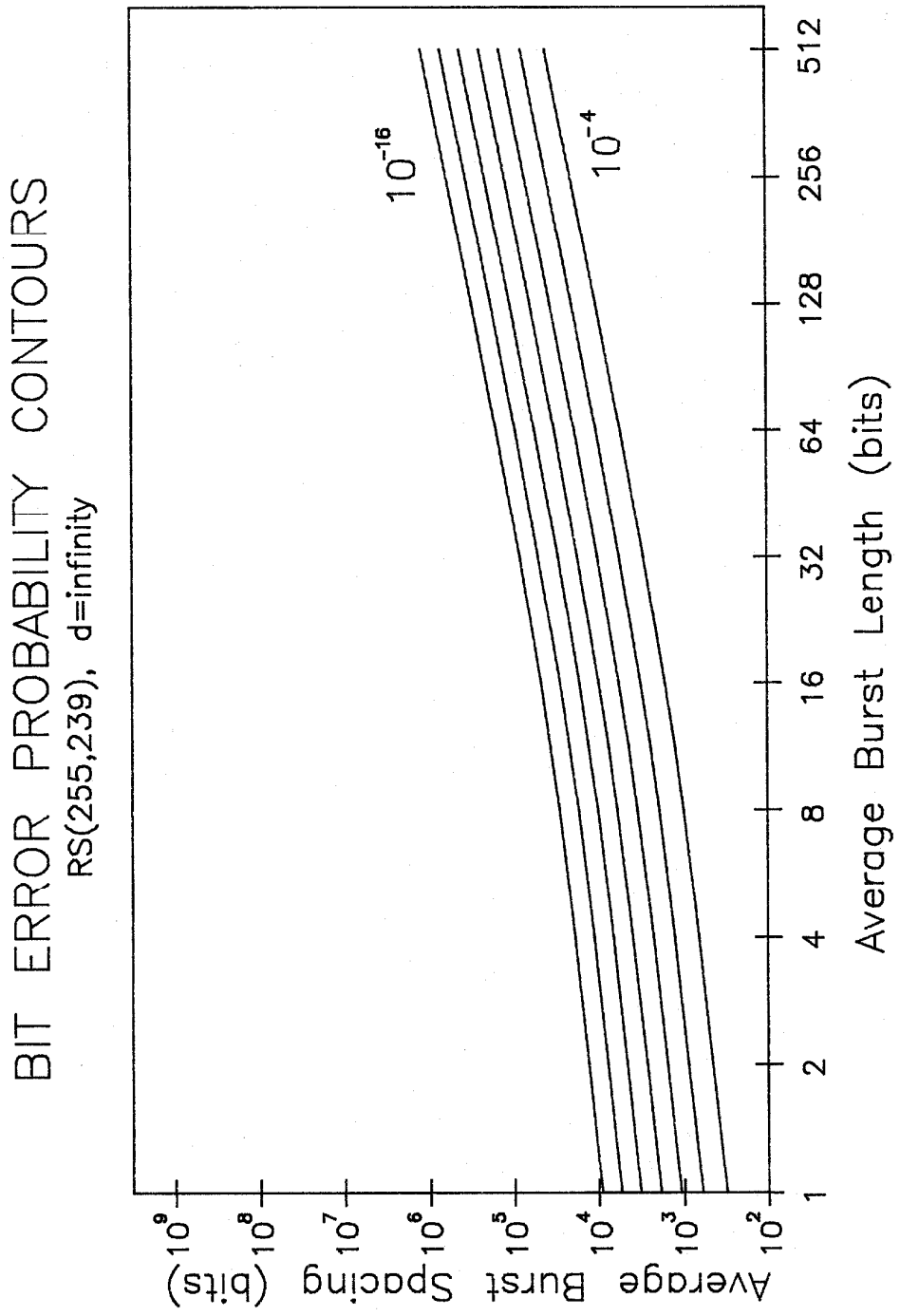

Figure B-14. Bit Error Probability Contours, $d = \infty$

Appendix C

Tables

The following pages contain several useful tables for dealing with finite fields. Table C-1, taken from Peterson [44], lists some irreducible polynomials of degree twelve or less. Octal notation is used for the binary coefficients. For example, in the list of polynomials of degree four, 23 corresponds to $x^4 + x + 1$. The letters following each polynomial signify:

A,B,C,D	Not Primitive
E,F,G,H	Primitive
A,B,E,F	Roots are linearly dependent
C,D,G,H	Roots are linearly independent
A,C,E,G	Reciprocal roots are linearly dependent
B,D,F,H	Reciprocal roots are linearly independent.

The reciprocal polynomial, which is also irreducible, has as roots the multiplicative inverses of roots of the given polynomial, and can be obtained by reversing the order of the coefficients. If the polynomial is primitive, so is the reciprocal polynomial. The first polynomial given for each degree is primitive; let us call one of its roots α. The integer k preceding all polynomials of that degree indicates that the given polynomial has α^k as a root. Thus, from the table, if α is a root of $x^4 + x + 1$, then α^3 is a root of $x^4 + x^3 + x^2 + x + 1$.

Table C-2, taken from Brillhart and Zierler [50], gives a complete list (up to reciprocals) of all irreducible trinomials of the form $x^n + x^k + 1$ over GF(2). Primitive polynomials are indicated by a value for k given in italics. Only values of n up to 100 are included here. Refer to the original paper for all $n \leq 1000$.

Table C-3 gives the prime factorization of $2^m - 1$, for $3 \leq m \leq 34$.

Log and antilog tables are given for GF(16) in table C-3, using hexadecimal notation for the field elements (e.g., $A = 1010 \mapsto \alpha^3 + \alpha$). Here $\alpha^4 + \alpha + 1 = 0$.

DEGREE	2	1	7H					
DEGREE	3	1	13F					
DEGREE	4	1	23F	3	37D			
DEGREE	5	1	45E	3	75G	5	67H	
DEGREE	6	1	103F	3	127B	5	147H	7	111A
		11	155E					
DEGREE	7	1	211E	3	217E	5	235E	7	367H
		9	277E	11	325G	13	203F	19	313H
		21	345G					
DEGREE	8	1	435E	3	567B	5	763D	7	551E
		9	675C	11	747H	13	453F	15	727D
		19	545E	21	613D	23	543F	25	433B
		27	477B	37	537F	43	703H	45	471A
DEGREE	9	1	1021E	3	1131E	5	1461G	7	1231A
		9	1423G	11	1055E	13	1167F	15	1541E
		17	1333F	19	1605G	21	1027A	23	1751E
		25	1743H	27	1617H	29	1553H	35	1401C
DEGREE	10	1	2011E	3	2017B	5	2415E	7	3771G
		9	2257B	11	2065A	13	2157F	15	2653B
DEGREE	11	1	4005E	3	4445E	5	4215E	7	4055E
		9	6015G	11	7413H	13	4143F	15	4563F
DEGREE	12	1	10123F	3	12133B	5	10115A	7	12153B
		9	11765A	11	15647E	13	12513B	15	13077B

Table C-1. Irreducible Polynomials over GF(2)

n	k	n	k
2	*1*	47	*5, 14, 20, 21*
3	*1*	49	*9, 12, 15, 22*
4	*1*	52	*3*, 7, *19, 21*
5	*2*	54	9, 21, 27
6	*1*, 3	55	7, *24*
7	*1, 3*	57	4, *7, 22*, 25
9	1, *4*	58	*19*
10	*3*	60	*1*, 9, *11*, 15, 17, 23
11	*2*	62	29
12	3, 5	63	*1, 5*, 11, 28, *31*
14	5	65	*18, 32*
15	*1, 4, 7*	66	3
17	*3, 5, 6*	68	*9, 33*
18	*3, 7, 9*	71	*6, 9, 18, 20, 35*
20	*3*, 5	73	*25, 28, 31*
21	*2*, 7	74	35
22	*1*	76	21
23	*5, 9*	79	*9, 19*
25	*3, 7*	81	*4, 16, 35*
28	1, *3, 9, 13*	84	5, 9, 11, *13*, 27, 35, 39
29	*2*	86	21
30	1, 9	87	*13*
31	*3, 6, 7, 13*	89	*38*
33	10, *13*	90	27
34	7	92	21
35	*2*	93	*2*
36	9, *11*, 15	94	*21*
39	*4, 8, 14*	95	*11,17*
41	*3, 20*	97	*6, 12, 33, 34*
42	7	98	*11, 27*
44	5	100	15, 19, 25, *37*, 49
46	1		

Table C-2. Irreducible Trinomials, $x^n + x^k + 1$, over GF(2)

m	Factorization of $2^m - 1$	m	Factorization of $2^m - 1$
3	7	19	524287
4	3 × 5	20	3 × 5 × 5 × 11 × 31 × 41
5	31	21	7 × 7 × 127 × 337
6	3 × 3 × 7	22	3 × 23 × 89 × 683
7	127	23	47 × 178481
8	3 × 5 × 17	24	3 × 3 × 5 × 7 × 13 × 17 × 241
9	7 × 73	25	31 × 601 × 1801
10	3 × 11 × 31	26	3 × 2731 × 8191
11	23 × 89	27	7 × 73 × 262657
12	3 × 3 × 5 × 7 × 13	28	3 × 5 × 29 × 43 × 113 × 127
13	8191	29	233 × 1103 × 2089
14	3 × 43 × 127	30	3 × 3 × 7 × 11 × 31 × 151 × 331
15	7 × 31 × 151	31	2147483047
16	3 × 5 × 17 × 257	32	3 × 5 × 17 × 257 × 65537
17	131071	33	7 × 23 × 89 × 599479
18	3 × 3 × 3 × 7 × 19 × 73	34	3 × 43691 × 131071

Table C-3. Prime Factorization of $2^m - 1$

i	α^i	$Tr(\alpha^i)$	α^j	j
0	1	0	1	0
1	2	0	2	1
2	4	0	3	4
3	8	1	4	2
4	3	0	5	8
5	6	0	6	5
6	C	1	7	10
7	B	1	8	3
8	5	0	9	14
9	A	1	A	9
10	7	0	B	7
11	E	1	C	6
12	F	1	D	13
13	D	1	E	11
14	9	1	F	12

Table C-4. Log/antilog Table for GF(16)

References

[1] Aho, A., J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachussetts, 1974.

[2] Benice, R. and A. Frey, Jr., "An Analysis of Retransmission Systems," *IEEE Transactions on Communication Technology*, **COM-12**, 135-145, 1964.

[3] Bequillard, A., B. Johnson, and S. Meehan, "Transform Decoding of Reed-Solomon Codes, Volume II: Logical Design and Implementation," ESD-TR-82-403, Vol. II (The Mitre Corporation), 1982.

[4] Berlekamp, E., *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.

[5] Berlekamp, E., "Bit-Serial Reed-Solomon Encoders," *IEEE Transactions on Information Theory*, **IT-28**, 869-874, 1982.

[6] Berlekamp, E., "The Construction of Fast, High-Rate, Soft Decision Block Decoders," *IEEE Transactions on Information Theory*, **IT-29**, 372-377, 1983.

[7] Berlekamp, E., "The Technology of Error-Correcting Codes," *Proceedings of the IEEE*, **68**, 564-592, 1980.

[8] Berlekamp, E., and L. Welch, "A New Reed-Solomon Decoding Algorithm," to be published.

[9] Blahut, R., "Transform Techniques for Error Control Codes," *IBM J. Res. Develop.*, **23**, 299-315, 1979.

[10] Blahut, R., *Theory and Practice of Error Control Codes*, Addison-Wesley, Reading, Massachusetts, 1983.

[11] Blahut, R., "A Universal Reed-Solomon Decoder," *IBM J. Res. Develop.*, **28**, 150-158, 1984.

[12] Bose, R., and D. Ray-Chaudhuri, "On a Class of Error Correcting Binary Group Codes," *Information and Control*, **3**, 68-79, 1960.

[13] Burton, H., "Inversionless Decoding of Binary BCH Codes," *IEEE Transactions on Information Theory*, **IT-17**, 464-466, 1971.

[14] Brent, R. and H. Kung, "Systolic VLSI Arrays for Polynomial GCD Computations," CMU Computer Science Department Report, Carnegie-Mellon University, 1982.

[15] Cain, J., and G. Clark, *Error-Correction Coding for Digital Communications*, Plenum

Press, New York, 1981.

[16] Cheng, U., "On the Continued Fraction and Berlekamp's Algorithm," *IEEE Transactions on Information Theory*, **IT-30**, 541-544, 1984.

[17] Chien, R., "Cyclic Decoding Procedures for BCH Codes," *IEEE Transactions on Information Theory*, **IT-10**, 357-363, 1964.

[18] Citron, T., Ph.D. Thesis, Stanford University, 1984.

[19] Cohen, Earl T., "On the Implementation of Reed-Solomon Decoders," Ph.D. Thesis, University of California, Berkeley, 1983.

[20] Consultative Committee for Space Data Systems, "Recommendation for Space Data System Standards: Telemetry Channel Coding," Jet Propulsion Laboratory, Issue 0, 1984.

[21] Conway, L. and C. Mead, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachussetts, 1980.

[22] Cyclotomics, Inc., "Interleaved Coding for Bursty Channels," SBIRC Report, 1983.

[23] Dally, W., and D. Whiting, "An Architecture for Systolic Reed-Solomon Encoders and Decoders," Caltech Computer Science Department Display File, #5145, 1984.

[24] El Gamal, A., J. Greene, K. Pang, "VLSI Complexity of Coding," Proceedings of 1984 Conference on Advanced Research in VLSI, Massachussetts Institute of Technology, 150-158, 1984.

[25] El Gamal, A., J. Greene, K. Pang, to be published in *IEEE Transactions on Information Theory*.

[26] Forney, D., "On Decoding BCH Codes," *IEEE Transactions on Information Theory*, **IT-11**, 549-557, 1965.

[27] Gilbert, E., "Capacity of a Burst-Noise Channel," *Bell System Technical Journal*, **39**, 1253-1265, 1960.

[28] Gorenstein, D. and N. Zierler, "A Class of Error-Correcting Codes in p^m Symbols," J. Soc. Ind. Appl. Math., **9**, 207-214, 1961.

[29] Herstein, I. N., *Topics in Algebra*, Wiley, New York, 1975.

[30] Hocquenghem, A., "Codes Correcteurs D'erreurs," *Chiffres*, **2**, 147-160, 1959.

[31] Deutsch, L., et al., "A Systolic VLSI Design of a Pipeline Reed-Solomon Decoder," JPL Technical Report, 1983.

[32] Deutsch, L., et al., "The VLSI Design of A Reed-Solomon Encoder Using Berlekamp's Bit-Serial Multiplier Algorithm," from *Proceedings of the Third Caltech Conference on VLSI*, 303-330, Computer Science Press, 1983.

[33] Knuth, D., *The Art of Computer Programming*, Vol. 3, Addison-Wesley, Reading, Massachusetts, 1973.

[34] Lidl, R., and H. Niederreiter, *Finite Fields*, Addison-Wesley, Reading, Massachusetts, 1983.

[35] Liu, T., "A New Decoding Algorithm for Reed-Solomon Codes," Ph.D. Thesis, University of Southern California, 1984.

[36] MacWilliams, J., and Sloane, N., *The Theory of Error-Correcting Codes*, North Holland Publishing Co., Amsterdam, 1977.

[37] Massey, J., "Shift-Register Synthesis and BCH Decoding," *IEEE Transactions on Information Theory*, **IT-15**, 122-127, 1969.

[38] Massey, J., unpublished lecture notes, 1983.

[39] Massey, J., and J. Omura, "Computational Method and Apparatus for Finite-Field Arithmetic," U.S. Patent Application #418039.

[40] McEliece, R., *The Theory of Information and Coding*, Addison-Wesley, Reading, Massachusetts, 1977.

[41] Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

[42] Murakami, H., I. Reed, and L. Welch, "A Transform Decoder for Reed-Solomon Codes in Multiple-User Communications Systems," *IEEE Transactions on Information Theory*, **IT-23**, 675-682, 1977.

[43] Peterson, W., "Encoding and Error-Correction Procedures for Bose-Chaudhuri Codes," *IEEE Transactions on Information Theory*, **IT-6**, 459-470, 1960.

[44] Peterson, W. and E. Weldon, *Error-Correcting Codes*, MIT Press, Cambridge, Massachusetts, 1972.

[45] Reed, I., and G. Solomon, "Polynomial Codes over Certain Finite Fields," *J. Soc. Indust. Appl. Math*, **8**, 300-304, 1960.

[46] Shannon, C., "A Mathematical Theory of Communication," *Bell System Technical Journal*, **27**, 379-423 and 623-656, 1949.

[47] Sugiyama, Y., et al., "A Method for Solving Key Equation for Decoding Goppa Codes," *Information and Control*, **27**, 87-89, 1975.

[48] Thompson, C., "Fourier Transforms in VLSI," *IEEE Transactions on Computers*, **C-32**, 1047-1057, 1983.

[49] Thompson, C., "The VLSI Complexity of Sorting," *IEEE Transactions on Computers*, **C-28**, 1171-1181, 1983.

[50] Zierler, N. and Brillhart, J., "On Primitive Trinomials (Mod 2)," *Information and Control*, **13**, 541-554, 1968.